CR-152061

UNCLASSIFIED

(NASA-CR-152061) NUMERICAL AERODYNAMIC SIMULATION FACILITY PRELIMINARY STUDY, VOLUME 1 Final Report (Burroughs Corp.) 133 p HC A07/MF A01 CSCL 14B

N78-10122

Unclas
G3/09 52520

FINAL REPORT

NUMERICAL AERODYNAMIC SIMULATION FACILITY

PRELIMINARY STUDY

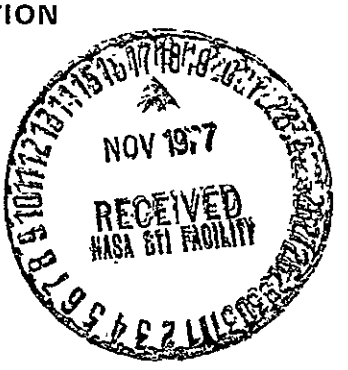
October 1977

VOLUME I

Prepared under Contract No. NAS2-9456 by
Burroughs Corporation
Paoli, Pa.

for

AMES RESEARCH CENTER
NATIONAL AERONAUTICS AND SPACE ADMINISTRATION



UNCLASSIFIED

CR-152061

UNCLASSIFIED

FINAL REPORT

NUMERICAL AERODYNAMIC SIMULATION FACILITY

PRELIMINARY STUDY

October 1977

VOLUME I

Prepared under Contract No. NAS2-9456 by
Burroughs Corporation
Paoli, Pa.

for

AMES RESEARCH CENTER
NATIONAL AERONAUTICS AND SPACE ADMINISTRATION

UNCLASSIFIED

CONTENTS
VOLUME I

CONTENTS (Cont'd)

CONTENTS (Cont'd)

CONTENTS
VOLUME II

CONTENTS (Cont'd)

CONTENTS (Cont'd)

LIST OF ILLUSTRATIONS

LIST OF ILLUSTRATIONS (Cont'd)

LIST OF TABLES

INTRODUCTION

Burroughs Corporation is pleased to submit this final report of the findings of the Numerical Aerodynamic Simulation Facility (NASF) Preliminary Study. This report presents a unique solution to the problem of numeric aerodynamic simulation. The solution consists of a computing system designed to meet the stated objective of providing an effective throughput of one billion floating point operations per second for three dimensional Navier-Stokes codes. Burroughs presents this design with full confidence that it is feasible to complete the detailed design and construction of this machine within the required time-frame. This high level of confidence is based on Burroughs' extensive and continuing experience in the design and development of very high performance computer systems. It is Burroughs' belief that the computer industry will not produce a commercial general purpose machine with the required performance by the early 1980's. Consequently, we feel that the design and construction of a relatively specialized system is not only feasible, but necessary to the achievement of NASF objectives.

This view is based on two business judegments. First, projections of both computing-power and cost of performance of commercial computers for the 1980 to 1985 time-frame do not include a machine of this capacity or price. That is, a generation gap will exist between any NASF implementation and concurrent commercial products. Second, market trends indicate that an insufficient market

exists to sustain development of a machine with two orders of magnitude speed increase on a commercial basis.

In summary, we believe that the system presented in this report constitutes the best approach to meeting the NASF goals in a timely and cost-effective manner, and that NASA has an opportunity to maintain a "forefront" position in the scientific community while achieving these goals.

This report consists of two volumes. Volume 1 provides an overview of the NASF preliminary study, and presents the results. Volume 2 contains additional technical data in support of these results, including detailed discussion of key design issues.

Volume 1 is organized into four chapters.

Chapter 1	presents an overview of the study. The organization and methodology are described to identify the origin and development of the results.
Chapter 2	presents an overview of the NASF system, emphasizing the Navier-Stokes Solver (NSS) which is the key element.
Chapter 3	is a detailed discussion of the hardware of the NSS.
Chapter 4	presents a discussion of the software elements of the NASF.

CHAPTER 1

NASF STUDY OVERVIEW

The results of this study have produced a unique solution to the problem of numeric aerodynamic simulation for three-dimensional Navier Stokes equations. In order to fully appreciate the design, its features, and subtleties, the methodology of the study which evolved this solution must be understood. This chapter is intended to explain that methodology. First, the problem and solution, in brief, will be presented, then basics of the study approach will be explained. Next, a description of each of three sub-studies follows with emphasis on specifically what was examined and why. Finally, the results of the sub-studies are merged to highlight their impact on the processor architecture evolution, and show how the "baseline design" for NASF was selected. Subsequent chapters will discuss details of that design.

STUDY OBJECTIVES

The Numerical Aerodynamic Simulation Facility Preliminary Study Objectives were to determine the feasibility of designing a system delivering one billion floating point operations per second effective throughput for three dimensional Navier Stokes codes by 1982. If feasible, a processor architecture and functional design definition were to be developed, supporting that assertion, with attendant requirements of power, size, cost, schedule, etc.

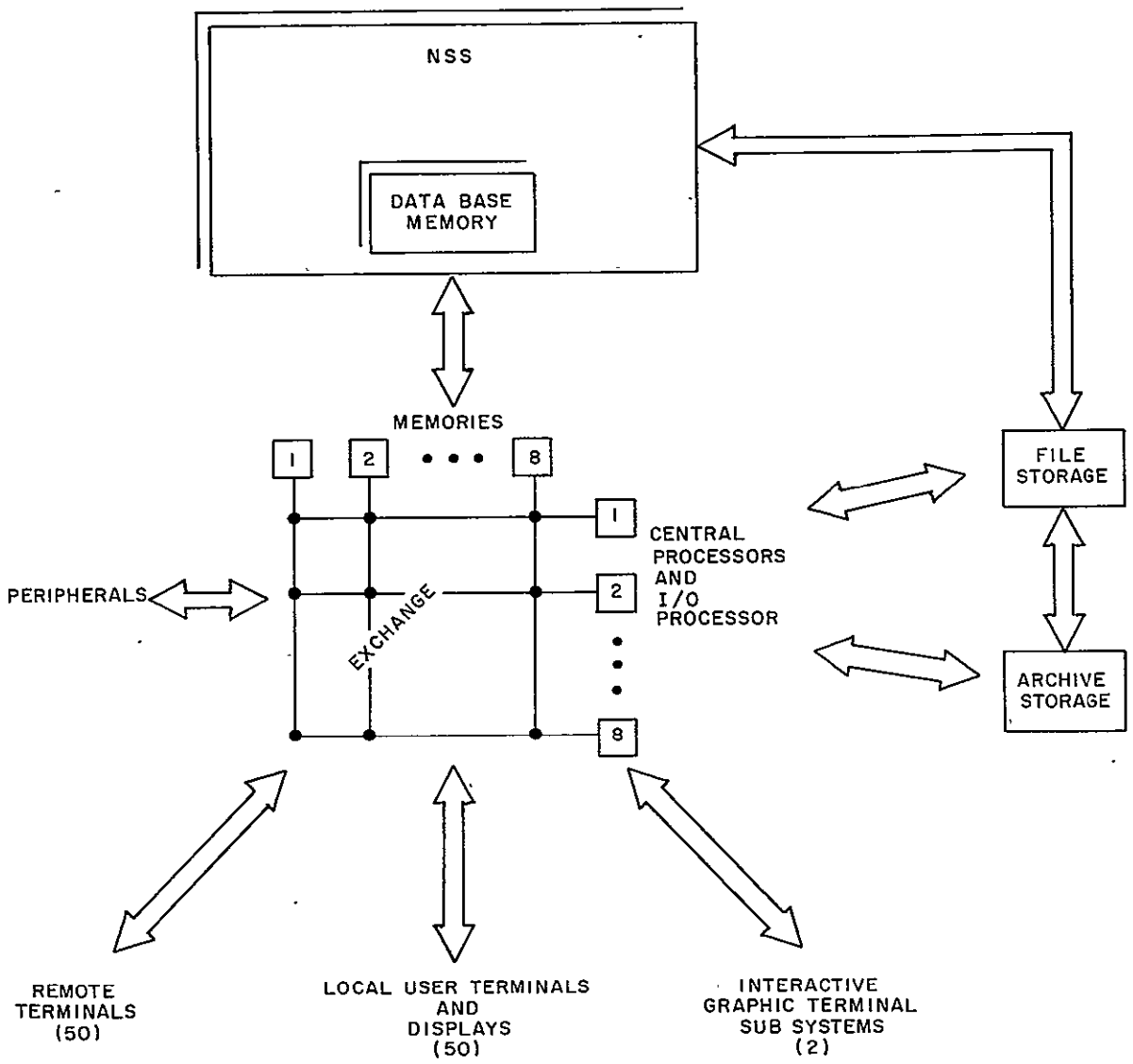


Figure 1-1. NASF System Block Diagram

1.1 NASF OVERVIEW

The basic structure of the candidate baseline NASF system is shown in Figure 1-1. The major elements are:

- The Host, a Burroughs B 7800 multiprocessing system
- An Archival Storage system
- File Memory
- The Navier-Stokes Solver (NSS) ... the high throughput workhorse of the system.

The Host Computer

The host, a Burroughs B 7800 system, acts as the system manager and support facility. It provides the user interface, schedules and dispatches NSS tasks, and executes supporting functions such as compilation, data reduction, and output generation.

The Navier Stokes Solver (NSS)

The NSS is the high throughput computational element. It is a highly parallel processing array, designed to provide the required computational throughput on three-dimensional Navier Stokes programs. The Data Base Memory (DBM) of the NSS provides the interface between the NSS and other system elements. The program and data files are loaded to the DBM by the host. The NSS with the DBM constitute a high speed "computational envelope", allowing the NSS to run at maximum speed essentially without outside interruption or dependence until job completion.

The Archive Memory

The Archive provides a very large storage capability for long term retention of programs and data bases. It consists of a commercially available mass memory system, which is managed by the host.

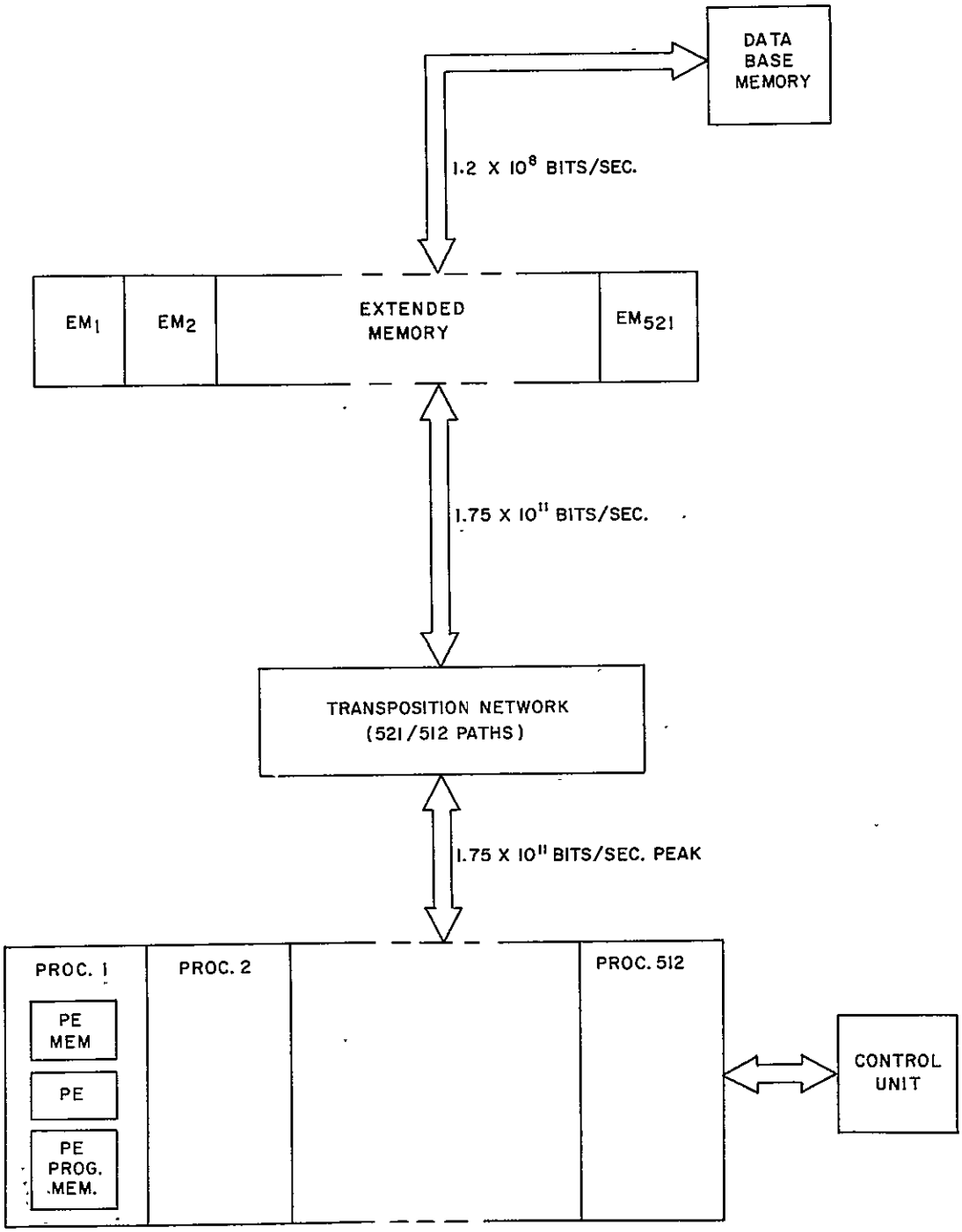


Figure 1-2. SAM Block Diagram

The File Memory (FM)

The FM provides for short term file retention, staging and buffering between the host, the archive, and the DBM. It consists of a standard disk pack sub-system, and is also managed by the host.

1.1.1 Major Elements of the Navier Stokes Solver

The principal innovation in the NASF system is the NSS. The organization of the NSS is shown in Figure 1-2, and its characteristics are summarized in Table 1-1. The major features of this processing array are:

- Highly Parallel Architecture

  The NSS consistsof 512 computational processors, each with its own local data and program memories. These are coordinated by a single control unit, and connected via a transposition network to 521 modules of extended memory.

- Synchronizable Operation

  This feature of the NSS suggests the name we have given to the computational array, the Synchronizable Array Machine, or SAM. Previous processor arrays have operated in "LOCKSTEP", essentially synchronizing on every instruction cycle. The computational array of the NSS is synchronized explicitly by the code stream only when necessary. Between synchronization points, the individual processing elements may operate asynchronously, allowing them a degree of freedom in scheduling instruction sequences.

- Conflict Free Memory Access

  The transposition network between the processing elements and extended memory allows conflict free access to vectors in any dimension at full memory bandwidth. This eliminates the non-productive time which would otherwise be consumed by reordering or transposition of data before processing.

- Data Base Memory

  The Data Base Memory is the interface between NSS and host. By having all input and output restricted to this single point, the rest of the NSS is independent, and can proceed at full computational speed without any need for interaction with the host.

TABLE 1 NSS CHARACTERISTICS

Computational Capacity (On instruction mix)	1.7×10^9 floating operations/sec.		
Number of Processing Elements Number of Extended Memory Modules	512 521		
Memory capacities (total) Extended memory Processing element memories Processing element program memories	 34 million words 8 million words 4 million words		
Transfer rates (bits/sec)	per path	no. paths	total
PE - PEM	490×10^6	512	2.5×10^{11}
PE - PEPM	490×10^6	512	2.5×10^{11}
PE - (PEM+PEPM)	10^9	512	5×10^{11}
EM - via TN --PEM			
streaming mode	4×10^8	512	2×10^{11}
1 word/transfer	1×10^8	512	5.5×10^{10}
EM - DBM	—	—	1.4×10^8
Program loading to all PE's simultaneously			4×10^8 per PE
Clock, synchronous throughout the NSS	50 MHz minor cycles 25 MHz major cycles		
Total No. of IC packages, including memory (almost all LSI)	200,000		
Word Size:	48 Bits		

- System Balance

  All transfer rates and execution speeds are tuned to one another in concert with the requirements of the application. This provides for high efficiency by balancing the utilization of system elements.

- Ease of Use

  A high level user language, complemented by an instruction set oriented to efficient implementation of high level language programs allows ready access to the computational power of the NSS, without encumbering the user with assembly language programming or implementation details.

1.2 STUDY METHODOLOGY

Experience in the design and manufacture of data processing equipment, especially very-high-performance computer systems, leaves many lessons behind. In addition to knowing what a design team should do, there are some lessons about what should not be done.

The Burroughs study team took care to avoid a serious problem that often traps those aiming at maximum speed - namely pushing the state of the art on too many frontiers. One could rely on significant advances in

(1) Architecture,
(2) Hardware Technology, or
(3) Software Technology.

For increased performance we chose Advanced Architecture taking care to build on mature or developed software whenever possible. In addition hardware implementation will be conservative, consistent with performance goals, and will not rely on imposing inordinate speed requirements or new, untried technologies.

Selecting architectural elegance as the new frontier, the study concentrated on matching the architecture to the problem. Existing computer structures were not integrated to force-fit a "super-structure" of these units to the problem. The reasons were:

(1) Lack of Architectural Flexibility
(2) Inefficient and Not-Cost-Effective.

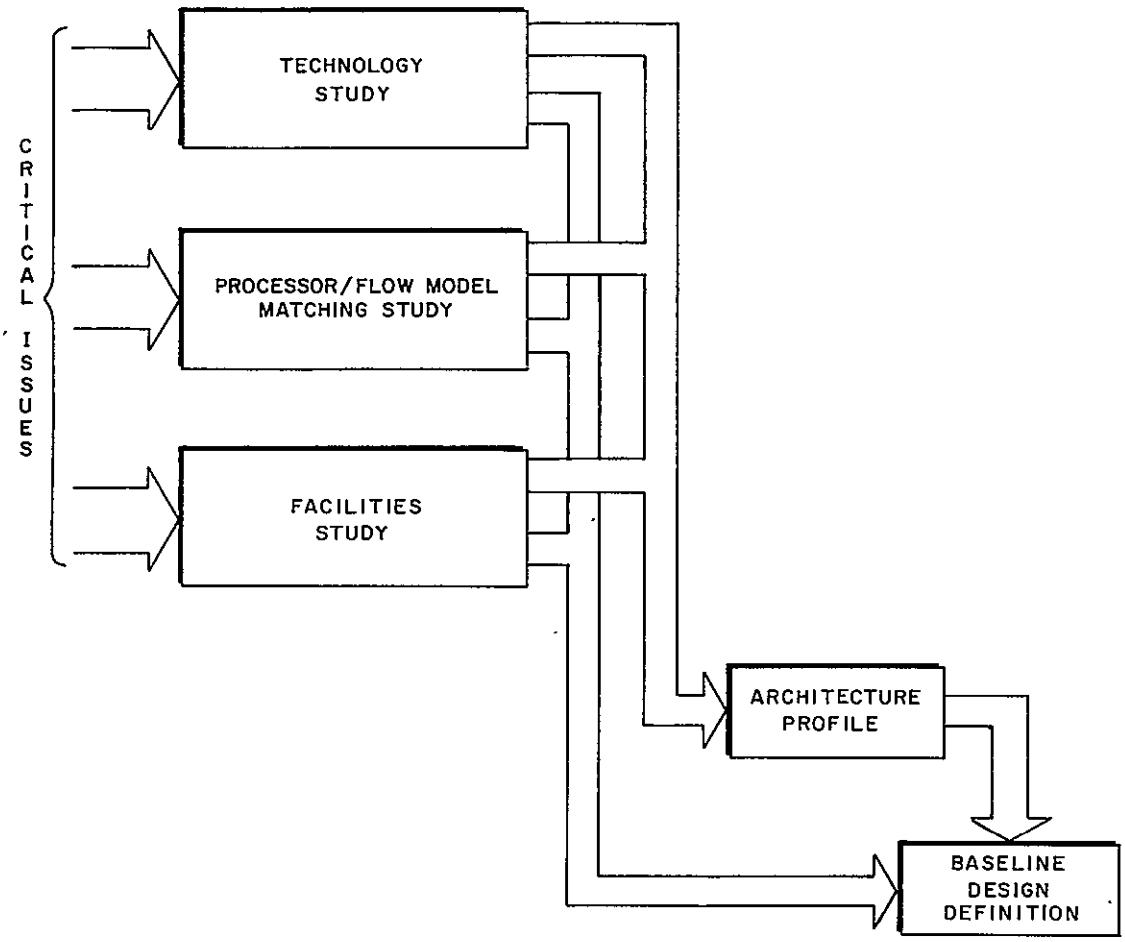


Figure 1-3. NASF Study Approach

Although performance requirements may be met in this fashion, the lack of architectural freedom with the structures implies that many hardware and software elements are not utilized, others must be customized resulting in a machine that has some "dead-wood".

The NASF system presented here was developed by evolution from careful analysis of the problem characteristics to insure a genuine fit. Top Down design fundamentals were practiced so that on each of the several design iterations, results could be traced to assumptions. Traceability of this sort allows bottlenecks or errors found to be identified at their origin where viable alternatives could be reexamined.

1.2.1 Sub-Studies

Specifically, three sub-studies were executed simultaneously as required by the original contract statement of work.

(1) The Technology Study developed a data base of logic and memory technologies by literature seraches, vendor interviews and conferences, etc. Trends of critical issues and parameters of these technologies were studies and a technology forecast developed for the 1980-1985 time frame.

(2) The Matching Study analyses the flow models and their characteristics and matched them against candidate processor architectures.

(3) The Facility Study established metrics for the total facility and, at a more detailed level, the facility issues addressing "buildability" of the final system.

Each sub-study was executed with two objectives as shown in Figure 1-3.

(1) How do results affect processor architecture choice?
(2) How do results affect specific design choices in the baseline design?

That is, first a processor architecture was evolved as a result of the sub-studies, then a second iteration of the studies supported a more detailed design to the functional design level referred to as the Baseline Design. The result is an NASF definition that directly addresses the salient issues of the problem itself. This NASF definition meets or exceeds all requirements and can be built with a high degree of confidence - an assertion of great significance for such an ambitious task.

1.3 TECHNOLOGY STUDY OVERVIEW

The objective of this phase of the study was to establish a technology forecast for the NASF time frame and assess which logic and memory technologies are most appropriate for the design of such a facility.

The approach taken consisted of the following four tasks:

(1) Data Gathering
(2) Establish Critical Issues
(3) Examine Technologies & Trends
(4) Extrapolate 1980-85 Forecast.

Data gathering consisted of a three phase effort: a comprehensive literature serach, trade conferences and workshops; and interviews with vendors and suppliers such as Motorola, Fairchild, National Semiconductor, Intel, Signetics, and Texas Instruments. The data base acquired is reviewed and analyzed in Chapter 5 (Vol. 2) of this report.

The critical issues which were established were of two types - those affecting performance and those affecting development.

Performance	Development
(1) Speed	(1) Cost
(2) Reliability	(2) Maturity
(3) Power	(3) Extensiveness
	(4) Availability

Metrics for judgement of these issues and clarifications of their importance were then developed and used as criteria in the architecture/design process.

Under performance issues, speed of a logic family may be judged by propagation delay times, while with memory the key figures are read/write times. Density refers to the average number of gates or memory cells per chip. Reliability is largely a function of density since failures frequently occur at the substrate to pin connection, and as the number of pin connections decreases per given function,

the reliability increases. Power consumption is a measure of the energy costs associated with a device. A smaller speed-power product indicates better system performance per kilowatt.

As to developmental issues, cost should be considered in the light of performance per dollar, as well as absolute cost. Maturity is determined by field verification of manufacturer's specification. Another consideration in selecting a technology is the availability of the devices. In addition, multiple sources for all componentry are essential. These factors are important considerations in the selection of a technology family.

The technology survey provided inputs to the study not only in the obvious area of surveying the implementation of digital logic, but also in some areas of packaging, random access and serial memories, and archives.

From the many technologies used to implement digital logic, which are discussed in Chapter 5 of Volume II, three are of sufficient interest to report here:

(1) ECL has been the technology of choice in implementing high-speed digital computers for over ten years; however, the speed-power product, and hence the amount of processing that can be done per watt of power, has been continually improved, and in the last year some LSI has been available in ECL. ECL is a mature but still developing technology, exemplified by Fairchild's "100K" ECL family. This family could be used as a starting point for a baseline design.

(2) I^2L has much better speed-power product than ECL, allowing far more functions per watt. It is currently too slow for the NASF requirement but both speed and availability of standard parts are improving each year. I^2L would consume considerably less power than ECL, and is currently utilized internally in LSI chips where the speed is tolerable.

(3) MESFETs promise another improvement, by an order of magnitude, in the speed-power product as compared to I^2L. They are also very fast; however, they are still in early development. Years of development will be required before the MESFET's technology becomes mature.

From this study we conclude that ECL is the most feasible current technology for implementation of an NASF design, and the base line design will begin with ECL as a starting point.

Memory technology represents an area of low risk for the NSS. 16K-bit dynamic RAM's (Random Access Memory) are currently available. 16K-bit static RAMS and 64K-bit dynamic RAMS are on the drawing board.

CCD shift register memory is currently available in pilot quantities in the 64K bits size. Another factor of four in storage size (256K-bits) is expected by 1980.

Manufacturers reported the occurrence of spontaneous errors in CCD memories. This leads to a requirement for continuously monitoring the contents of a CCD memory and rewriting it correctly when bit errors occur.

Present bubble memories put severe complexities into the controlling and driving circuitry, making them very difficult to use.

Sufficient information about the magnetic storages available for the archive was obtained to indicate that there are several commercially available contenders for the archive storage. No effort was made to determine which of today's contenders were likely to be withdrawn from the market in the next two years, nor to uncover the new contenders which are undoubtedly under development.

1.4 PROCESSOR - FLOW MODEL MATCHING STUDY OVERVIEW

The key sub-study in this effort was the Matching Study. Certainly, it had the most profound effect on the evolution of SAM as the chosen processor architecture as well as some design details. This sub-study was broken into several tasks prior to the actual matching or evolving process itself.

- Cataloging and examination of pertinent generic architectures for consideration to be used as a starting point.
- Establishment and discussion with NASA-Ames of critical issues and basic requirements and capabilities imposed on the architecture by the problem definition.

- Research and discussion of the fundamental characteristics of the flow models which affect the processor architecture.

Following these tasks, the results were merged with those of the other two sub-studies the total implications of which determined the final architecture.

Generic Architectures considered as starting points were:

- Hybrid system composed of analog computation devices with digital control and storage.
- Parallel array architectures with replicated arithmetic units executing the same program on different data achieving performance as a multiple of the number of arithmetic units.
  - Type 1 - Lock-Step synchronous arrays with clock-by-clock tight coupling of arithmetic units.
  - Type 2 - Non Lock-Step array with coupling at predetermined synchronization points rather than every clock
- Pipeline architectures where operations are streamed through different stages with performance as a multiple of the number of states.

A complete discussion of these generic architectures is found in Appendix L of the Final Report.

Critical issues, basic requirements and capabilities were jointly developed between the study team and NASA Ames personnel. Topics examined were:

(1) Navier Stokes Solver Capabilities

(2) Programming

(3) NSS - I/O

1.4.1 NSS Capabilities

The NSS is to solve the three-dimensional Reynolds averaged Navier Stokes equations, using both explicit/implicit and totally implicit, dimensionally-split, finite-difference methods.

The NSS is to compute, at high efficiency, problems containing a variety of boundary conditions which include the independent variables, their derivations, and other auxiliary variables, a variety of internal and external geometrics and a variety of turbulence models ranging from algebraic to 7 differential equation descriptions.

The NSS is to compute solutions for up to one million grid points. This implies a data base range from 14 million words for

5 conservation variables at 2 time levels
1 turbulence variable
3 grid coordinates

to 40 million words for

5 conservation variables at 2 time levels
7 turbulence variables at 2 time levels
3 grid coordinates
12 metrics (including time)
1 Jacobian

The NSS is to obtain steady state solutions for one million grid points in 10 minutes of CPU time for 3-D problems using algebraic turbulence models. At present this must be measured using 2-D explicit/implicit and implicit codes as performance metrics.

Two examples of typical programs and their computational requirements are given below:

Explicit code (MacCormack) status: A 2-D airfoil steady-state solution was obtained in 7 minutes on CDC 7600 for 2100 grid points. The steady-state was reached after 13 chord lengths of travel by computing inviscid solution for 7 chords and viscous solution for remaining 6 chords. Effective computing speed on 7600 is about 2 MFLOPS. Assuming twice the computational effort at each grid point for the 3-D case, this implies that to compute 13 chords in 10 minutes for one million grid points requires an effective computing speed of 1.4 gigaflops. Greater efficiencies by 1980 can be expected.

Implicit (Lomax, Steger) code status: A 2-D airfoil steady-state (13 chords traveled) was obtained in 10 minutes on CDC 7600 for 2300 grid points - all calculations were viscous. The effective computing speed on 7600 is about 2 gigaflops. This code implies that an effective computing speed of 2 gigaflops will be needed for a 3-D calculation over one million grid points. However, researchers working on the implicit code are confident that improvements in the treatment of boundary conditions and other strategies can improve the speed of the method by a factor of 2 which implies that at least a one gigaflop effective rate will be needed.

It is concluded that the minimum effective computing rate needed for the Navier Stokes problem is one gigaflop.

A precision of 10 decimal digits is required.

A general purpose computing capability equal to that of the highest speed commercially available computers of the early 1980's time frame is desired.

1.4.3 Programming

A high level programming language consistent with ease of mapping the solution methods onto the machine, optimum machine performance and the available language development time is necessary.

Desirable programmability features of the Navier-Stokes machine are as follows: A FORTRAN-like high level language with extensions necessary for efficient problem mapping, as well as the following features:

- a stable optimizing compiler
- good compiler diagnostics
- warning from the compiler of possible run-time inefficiencies
- ability to give good run-time diagnostics and statistics
- vector length independence
- freedom from the need to do explicit-mode vector manipulation
- ease in specifying data allocation

1.4.4 NSS I/O

The primary I/O activities of the machine are the input of initial problem parameters, restart from stored data, and the output of snap-shots and restart dumps. Another important activity is the output of debug dumps. Two basic types of Navier Stokes solutions are desired - steady and unsteady (or more correctly quasi-steady). Steady cases are characterized by the appearance of a solution that does not vary with time after some large number of time steps or large number of characteristic body lengths travelled. Unsteady cases are characterized by the appearance of a solution that is periodic in time after some large number of time steps. In order to analyze the unsteady or periodic nature of these solutions more time steps (on the order of six times that of steady cases) are required. Additional data output is also required in these cases. It is estimated that 75 percent of the time will be used to solve the steady flow case and the remaining 25 percent, the unsteady.

The following output capabilities for these cases are desired.

(1) Snap Shots

a. Integrated quantities such as drag, lift and moments approximately every 15-30 seconds.

b. Surface quantities such as pressure and skin friction. If the grid moves with time, the grid coordinates must also be output. A given quantity such as pressure, plus the coordinates could total up to approximately 60,000 words of output every 15-30 seconds.

c. Flow quantities in the field such as pressure or Mach number. For a grid of 1,000,000 points an entire field of, say, Mach numbers plus coordinates would be 4,000,000 words. However, it is anticipated that only selected grid points need to be output and this would be about one one-hundredth of the above of 40,000 words every 30 seconds. These snapshots require the heaviest output and for 60 minute runs would accumulate up to 50,000,000 words for the unsteady cases.

(2) Restart Dumps

(3) Debug Dumps

(4) Formatted I/O

1.4.5 Flow Model Code Characterization and Analysis

Codes supplied to us by NASA Ames were analyzed statically and dynamically to determine what the specific characteristics of the Flow Model problems are and how do they impact computer architecture. The codes studies were written for two specific computers. Features in each code that were specific to its target machine were stripped away to find the basic issues. The areas that were examined, group themselves naturally into those issues which address processor requirements, memory requirements, or communications requirements, and are outlined below.

Memory Requirements

(1) Data Base Size - (The actual input/output variables)

(2) Program Size

(3) Workspace Size (Those variables never outputted in normal production code - the temporaries)

(4) Access Patterns (dimensionality of problems, subarray structure, indexing patterns)

Communications between Processors and Memories

(1) Number of Computations per Data Base Access

(2) Interaction of Problem Variables

(3) Data Dependency

(4) Control Structures

(5) Access Patterns (planes, rows, columns, etc.)

Processor Requirements

(1) Word Size and Format

(2) Relative frequency of operations

(3) Index computations

(4) Number of input operands per output operands

(5) Scalar operations

(6) Frequency of intrinsics

(7) Program Structure

Each of these issues were examined in detail and the results are listed in Chapter 9 with a full discussion of the methodology. Several items were stated by NASA-Ames as required and were therefore assumed.

The study of the memory requirement showed that the canonical problem variables and number of grid points produce a data base memory of 14-40 million words (NASA-Ames requirement). The workspace size was found to be approximately 40 temporaries per database variable. This of course is programmer and architecture dependent and hence is only an indication of the relationship between work space and data base. It was found that the problem arrays are generally 4 dimensional with 3 geometric and 1 variable coordinate. They are accessed in a fairly regular manner in the sense that the indexing is a function of the loop variable plus or minus a small integer. There is almost no indexing that occurs as a function of loop variable and another integer variable set outside of the loop. The structure of the loops indicate that entire arrays are processed in a given piece of the computation rather than small subarrays. Program size is relatively small at under 4000 card images.

Requirements on communication between processor and memory structure were determined by a number of flow-model program parameters. The data dependency studies of variables in loops showed that there existed complex first order linear recurrences which were functions of each of the three geometric variables. These recurrences occurred in over 60 percent of the executing Implicit program. The study of the control or branching structures within the programs showed them to be relatively simple and generally linked to loop variables. Some were data dependent but when they occurred the variables were functions of inner loop parameters.

Further studies of the relationship between the data base memory requirements, the workspace requirement and the number of floating point operations showed that a fetch or store to data base memory occurred infrequently in comparison to the number of floating point operations. Typically the Implicit (Steger) program has an average incidence of 15 floating point operations per fetch.

Additionally, by investigation of the indexing patterns within loop structures one found that there is relatively low interaction among problem variables on different grid points. For example, variables are fetched from several adjacent points, computations are performed and then a result is stored relative to the grid point. There is no continual switching back and forth of index patterns. The access patterns appear to be simple rows, columns and planes with a skip distance of 1.

Processor requirement studies showed that multiply, add, and multiply-add instructions are extremely important floating point operations. For example in the Implicit Code it was found that 58 percent of all operations were multiplies, 44 percent were adds and 2.5 percent were divides. 60 percent of all operations occurred as multiply-add pairs. Division and intrinsics as SQRT and EXP occur rarely and double precision is never required. Since most of the array references are to three and four dimensional arrays, integer arithmetic calculations are a strong requirement. The combination of work space requirements and the average number of input operands to output operands (3.5) places certain requirements on the processor. NASA-Ames has additionally specified 10 digit accuracy requirement.

The data collected from the studies was used to define and delimit the characteristics of the requisite architecture. The output from the matching study together with the technology study and facilities study data were then used to develop definitions of an architecture discussed in Section 1.6.

1.5 FACILITY STUDY

The primary objectives of this sub-study were threefold:

- Identify housing and support requirements of the facility
- To provide cost and schedule engineering estimates for effective planning
- Assessment of NSS implementation issues as they would impact architecture and design choices.

These objectives were pursued by determining the facility requirements of units or subsystems already identified and placing reasonable bounds on facility requirements for those elements which have yet to be specified. After a preliminary definition of the NSS, an implementation schedule and an engineering cost estimate were assembled, and analyzed. As the NSS definition proceeded, additional iterations on the schedule and cost were performed.

Finally, the critical issues relevant to implementing the NSS were defined and guidelines developed to insure that the design would indeed be realizable. This effort raised some interesting considerations which impacted the architecture choice and some design details as well.

Critical issues affecting the implementation or realization of the NSS in particular are:

(1) Critical Path Analysis was examined to eliminate short waterfalls in the schedule by locating their source and minimizing their occurrence.

(2) Procurement problems can be avoided if there is an early identification of long-lead items, if custom componentry is minimized, if multiple sources are employed wherever possible, and if adequate protective documentation is obtained from each vendor. This issue can be the largest single risk factor in any program schedule, cost, and possibly performance.

(3) Production considerations include maximuzing the number of replicated units to minimize production learning curves and take advantage of economies of scale. Standardization of componentry, connectors, cables, etc., minimizes inventory problems and smoothes the production process.

(4) Module or Subsystem Interface Management demands the reduction of complexity of interconnections between all functional elements.

(5) Debugging and Maintenance: As in the production considerations, if the number of complex elements, which field engineers must work with, are kept to a minimum, then deubgging and maintenance are simplified - furthermore, this minimizes the inventory of spares.

(6) Packaging of any design must have the highest density consistent with heat removal. It must be such that the LRU (lowest replaceable unit) is easy to isolate, test and replace. Additionally, usage of common board types should be maximized.

(7) Logic Design Rules and Noise Budgets. A technology choice for the design must be mature enough to develop credible noise budgets, and provide adequate operational margins.

(8) Power. Finally, power considerations suggest that we avoid complex power distribution schemes, and concurrently maximize the distribution of heat dissipation. These considerations will lead to some interesting features explained in the next section.

1.6 ARCHITECTURE EVOLUTION

The selection of the Synchronizable Array Machine for the baseline system is here described as an evolution of concepts which grew out of the findings of the three substudies. A parallel architecture was selected after examination of 3 generic types, hybrid, pipeline, and parallel.

The hybrid was rejected for three reasons:

(1) Difficulty of Programming. Years have already been spent in algorithm research in digital form. Even more investigation would be needed to recast the aerodynamic flow equations into suitable form for analog computation.

(2) Inaccuracies, and Unpredictability of the Inaccuracy. Such limited accuracy as analog devices have is often data dependent, and changes with age as component values drift. In digital computation, any desired degree of accuracy can be specified.

(3) Untrustworthiness. Unlike a digital computation, where tests can continuously monitor the computation process to ensure that correct results are being produced, an analog computer is essentially open loop as far as error control is concerned. A faulty component or off-scale input produces an output voltage which is not distinguishable from the output voltage of a properly functioning component.

Although analog processors have a very high computation rate, these limitations make them totally unacceptable for the NASF.

Pipeline architectures as we know them today appear to suffer from inefficiencies, namely:

(1) Long start up times between vector operations,

(2) Difficulty in dealing with transpositions, and

(3) The need for massive amounts of work space memory to accommodate propagation of temporary variables.

Certainly these problems can be dealt with and solutions developed to make a pipeline a suitable architecture (as we have done for the parallel architecture) but a re-examination of the Facilities Study highlighted other issues which made the selection of a parallel array more sensible for Burroughs.

Assuming both architectures could be evolved to produce a design of equal performance, Burroughs is more confident that the parallel machine can be manufactured with less risk. The claim is based on observations:

(1) The large number of replicated units in a parallel array minimizes production and debugging and field engineering learning curves. Certain economies of scale could be realized in development as well.

(2) Burroughs experience, in three generations of parallel high performance systems (namely ILLIAC, PEPE, and the Burroughs Scientific Processor (BSP), provides an invaluable data base of knowledge in the detailed design and manufacture of such a system.

The beginning of the architectural development, therefore, was based on the generic parallel configuration shown in Figure 1-4.

From this point, the definition of SAM can be well understood as series of refinements based on results of the sub-studies.

The Alternating Direction Implicit method for solving aerodynamic flow models, with split operators, requires that data arrays be transposed during access. Planes are required to be fetched in parallel from any 2 of 3-dimensions in the same grid. This implies the need for an efficient transposition mechanism.

Several different designs were considered. The selected Transposition Network (TN) is a unique innovation offering:

(1) low parts count
(2) minimal data access delay
(3) simple control requirements
(4) simple but flexible data allocation

This design demands that memory be partitioned into a prime number of banks larger than the number of processors.

The Transposition Network (TN) is shown in Figure 1-5 as the first refinement of the generic parallel configuration.

The occurrence of a significant number of floating point operations to fetches (especially in the implicit code) implies a large workspace requirement. In fact up to 40 temporary variables per data base variable may be generated. Propagation of such a large number of temporaries throughout the machine would cause severe timing penalties. To mitigate this problem, local memories for each processor are required. In addition, the bandwidth of the TN can then be reduced without performance degradation. This makes the Transposition Network simpler and less costly. The absence of data dependencies among points in the same plane allows this refinement (Figure 1-6) to occur. The increased cost of many data memories in the processor is offset by the decreased requirement for storage capacity in Extended Memory for temporary variables. The nomenclature for the main memory can now be appreciated as Extended Memory (EM).

The result of this refinement allows one to think about parallelism as a series of vertical slices. That is:

Given a series of statements of the following form:

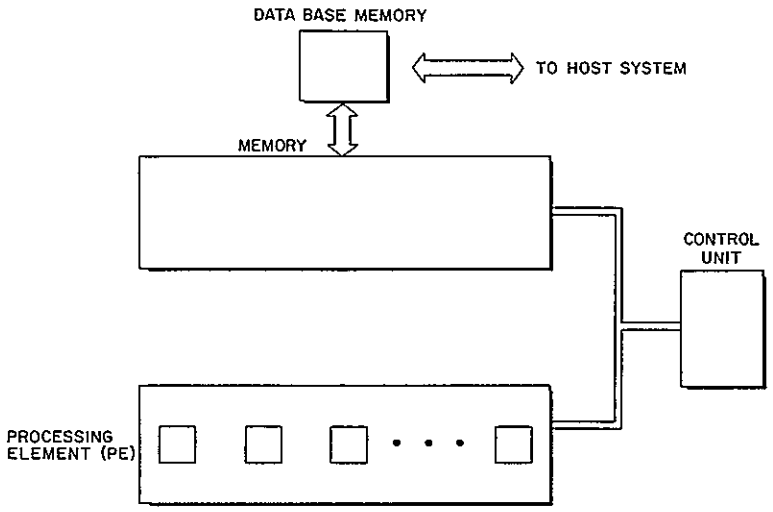


Figure 1-4. Parallel Configuration

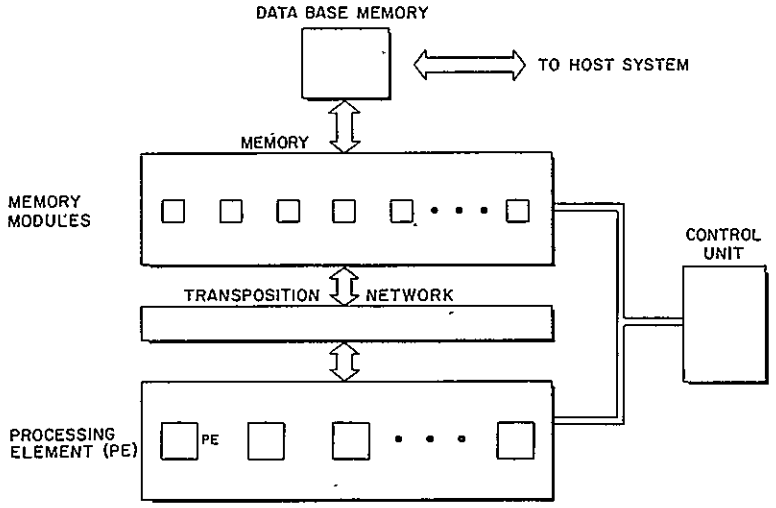


Figure 1-5. Parallel Configuration - Refinement 1

```
DOPARALLEL (on one or more indices, say I, J, K between limits)
STATEMENT 1 - involving variables indexed on the parallel indices
STATEMENT 2 - involving variables indexed on the parallel indices
     .
     .
     .
STATEMENT n - involving variables indexed on the parallel indices

ENDO
```

there are two ways of thinking of the parallelism.

In the first method, statement 1 is executed on the vectors implied by the parallel indices. Then statement 2 is executed as a vector statement, and so on up to the nth statement. Having each statement executed separately as a vector statement is called "horizontal slicing" of the parallelism.

The second method is to assign a processor to a particular instance of the set of indices. Processor 17, for example, may handle all computation associated with J=1 and K=19, while processor no. 222 handles J=3 and K=22. Each processor now executes, essentially independently, a piece of code involving the I index. This kind of parallelism has been called "vertical slicing". Vertical slicing is appropriate when, as in the Navier-Stokes equations, there is little interaction between the variables at one grid point and the variables at another.

Three or more generations of parallel processors have been shown that instruction interpretation of parallel constructs by the CU creates a bottleneck. The CU must be extremely fast to keep up with the array. Its complexity is severe enough without this responsibility. The program size has been observed to be small enough to consider placing program memories in each processor as shown in Figure 1-7. This now results in a stand alone processor with manageable interface (very few lines) to the control unit, elimination of massive cabling and a simpler CU. These savings and their attendant design, schedule issues will offset the cost of multiple copies of the program memory, as well as improve performance.

In a parallel array with a single program memory, the distribution of instructions by the CU serves to synchronize the operation of the PE's. Distribution of the program to local program memories results in a requirement for a synchronization

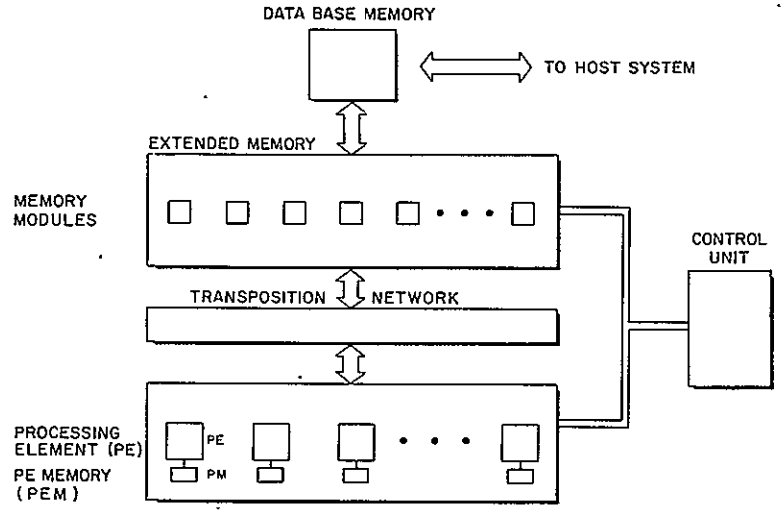


Figure 1-6. Parallel Configuration - Refinement 2

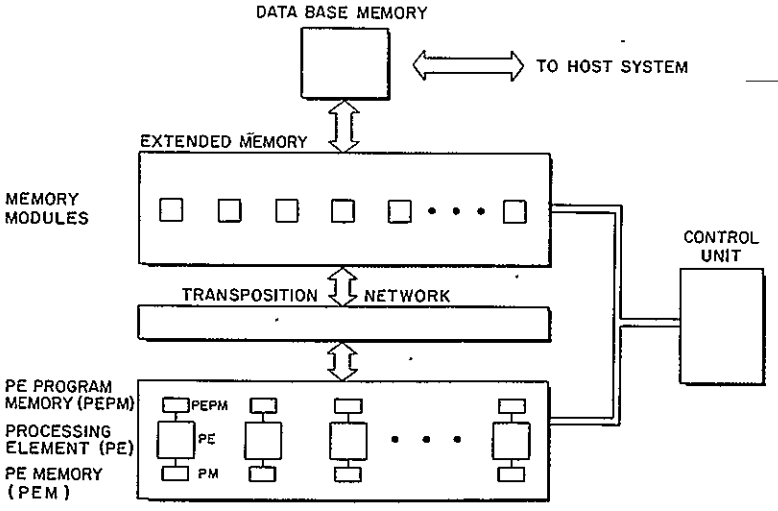


Figure 1-7. Parallel Configuration - Refinement 3

mechanism between CU and PE's. To provide maximum flexibility, we elected to invoke the synch mechanism explicitly in the code stream (Figure 1-8). This allows synchronization to occur only when necessary, (i. e., just prior to parallel fetches and stores). This allows data dependent branches to be run concurrently, and data dependent instruction options (e. g., round after normalize if overflow) to be executed only when needed. Since a limited level of concurrency of different operations can occur, idle processors can execute confidence checks on themselves. Different code sequences for boundary conditions or on either side of a shock front may be accommodated as well.

The choice of 512 as the number of processors is based primarily on the highest expected speed of efficient memory chips. 16k-bit static RAM chips are expected to be available at about 100 ns cycle time, by 1980, and are appropriate to PEM, PEPM, and Control Unit Memory (CUM). 64k-bit dynamic RAM chips are expected to be available at nearly the same time, at speeds nearly matching the present 200 ns or so speed of current 16k dynamic RAMs. These are the memory chips in the baseline system.

Consider, for example, the effect on the design of a choice of 256 processors. The twice-as-fast PEM and PEPM memories would require 50 ns chips, which would be available only in a 4k-bit size. Thus, the total number of memory chips would double, from the 37,888 memory chips of the baseline system to a total of 75,776 chips. The twice as fast EM would require 16K-bit chips to maintain the same speed, and its size would quadruple from 29,176 memory chips to 116,704 chips. Parts count in the twice-as-fast processor is estimated to double, making no net savings, but increasing the required design effort.

The size of data base for codes expected to execute for 10 minutes indicates as much as 10^{17} bits of data are operated upon. To expect no failures in that time is ambitious indeed, therefore it was necessary to impose a strict philosophy of fault detection and correction in the design of the hardware and software, including:

(1) Hardware Error Detection
(2) Hardware Error Correction
(3) Arithmetic Residue Checking

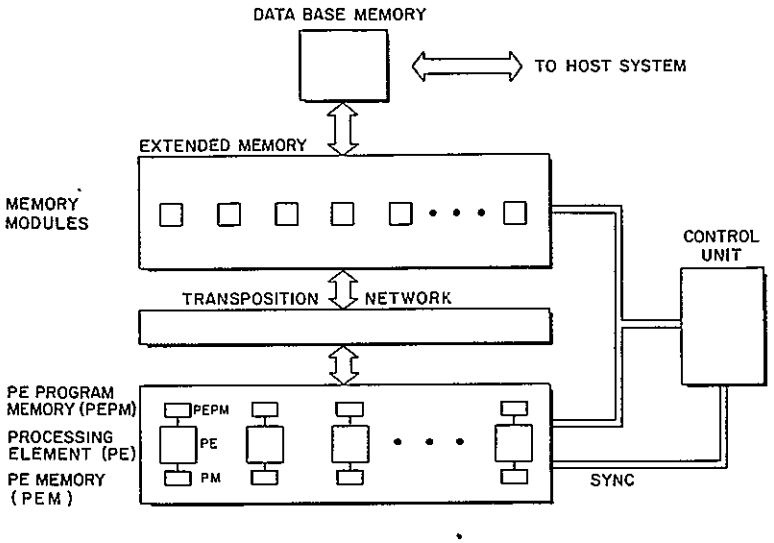


Figure 1-8. Parallel Configuration - Refinement 4

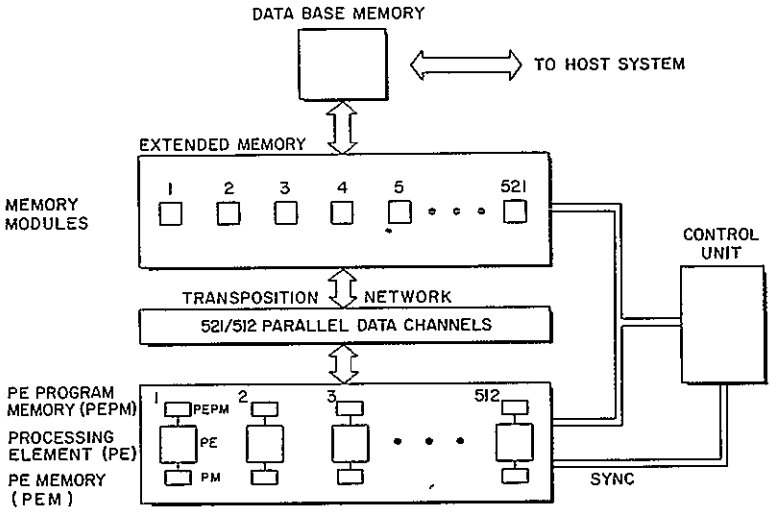


Figure 1-9. Parallel Configuration - Refinement 5

Figure 1-9 is a block diagram of SAM, the Baseline Design for the NSS. Its evolution, as well as subsequent design decisions and guidelines results in a design which features:

1.7 HIGH THROUGHPUT

The throughput potential of the NSSs is 1.7 billion floating point operations per second. This is derived from the relative ratios of the instruction mix combined with the expected execution time of the operations. This yields 294 nano sec/512 floating point operations which is equivalent to 1.7 Billion Floating Point Operations/sec. Additional study of the baseline for the specific codes indicates that the required effective rate of 1 Billion FLOPS is achievable.

1.8 EASE OF USE

High level language requirements, the guidelines of matching code to the user language and indeed the use of a High level language to write the compiler were important decisions made early in the study. The Vertical Slice Concept allows all classical serial optimization techniques to be utilized on the SAM. Recognizing that this architecture has unprecedented flexibility, it is incumbent upon the compiler to have debug aids to protect the user.

The protected environment in which SAM operates - the high speed computational envelope - isolated from the rest of the system requires that it have only a very small operating system of its own. I/O to and from that envelope will not be incumbrence upon the user or SAM as well.

This architecture is designed for a class of applications, yielding somewhat degraded performance for:

(1) Problems with intimate arithmetic data dependency from one grid point variable to another,

(2) Interactive environments, and

(3) Multiprogramming environments.

In conclusion, we feel this represents a unique solution to the problem of numeric aerodynamic simulation, and Burroughs presents this design with full confidence in its feasibility. We believe that this system is the best approach to meeting the NASF goals in a timely and cost-effective manner, maintaining NASA's position in the forefront of scientific endeavor.

CHAPTER 2

THE BASELINE SYSTEM, A SYNCHRONIZABLE ARRAY MACHINE

The following description is that of the baseline system, a specific design of a synchronizable array machine (SAM), which has been selected as suitable for the NASF. This design will undoubtedly be refined as the design effort progresses.

2.1 OVERVIEW

The Numerical Aerodynamic Simulation Facility (NASF) simulates aerodynamic conditions using the time-averaged Navier-Stokes equations. The equations are solved on a Navier-Stokes Solver (NSS) which has a nominal computational throughput of approximately 1.7 billion floating point numerical results per second and a sustained throughput on the actual algorithms of over one billion numerical results per second.

The design of the Navier-Stokes Solver has resulted from solutions to several design questions:

- Hardware of more than adequate computational power
- Efficient data flow. Elimination of non-productive moving of data from one location in memory to another.
- Programming methods that make efficient use of the hardware capabilities.
- Programming methods that are easy for the user to understand and apply.

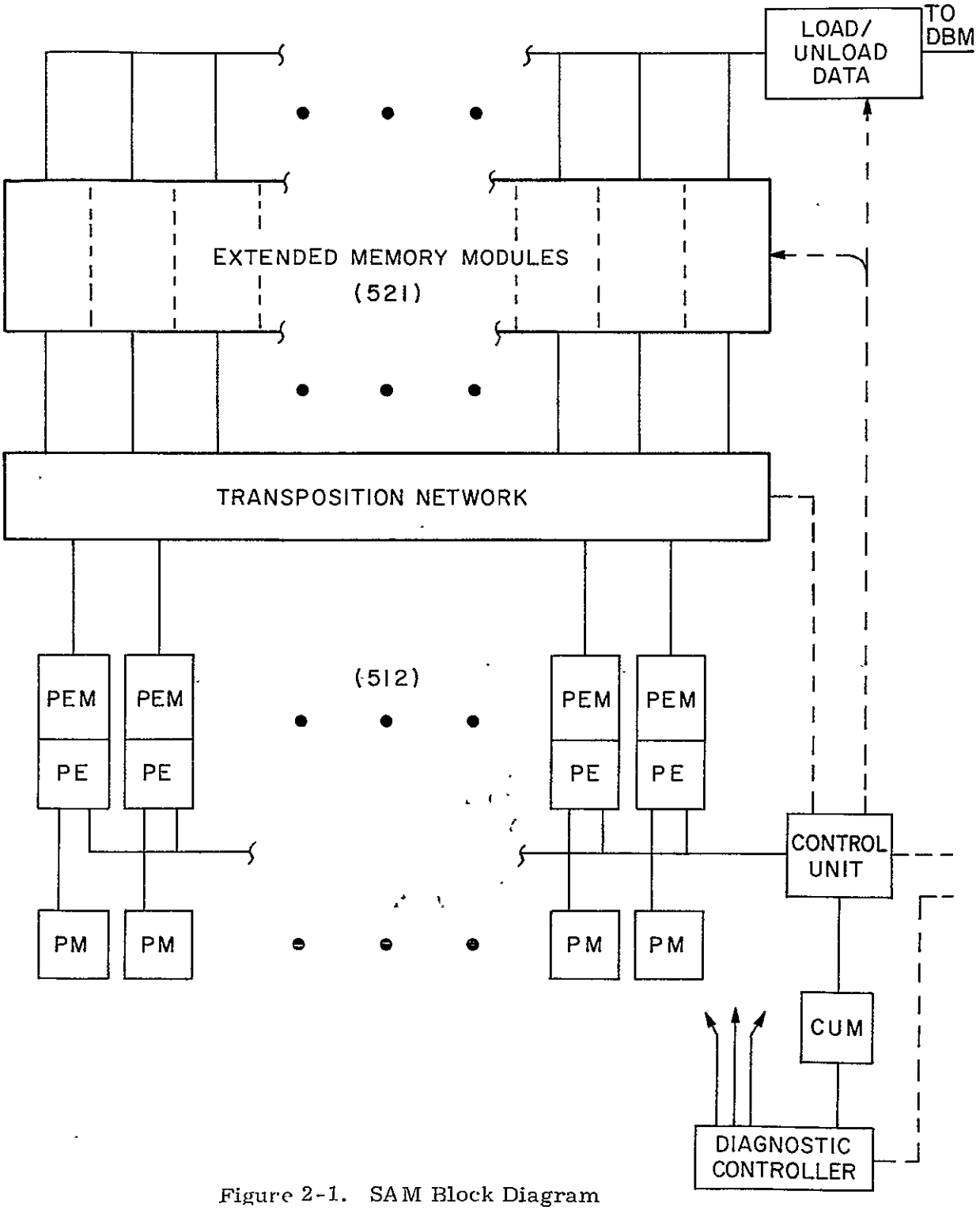


Figure 2-1. SAM Block Diagram

The NSS uses up-to-date technology for its hardware, and contains less than 200,000 integrated circuits including all logic and memory.

Reliability and trustworthiness of results are guaranteed by a number of stratagems which are described herein.

The Navier-Stokes Solver is embedded in a system that uses a commercially available processor as the host computer. Figure 2-1 is a block diagram of that system. Details of the host processor are not different from those of many existing installations. The file memory is made of standard disk packs which are the standard product line disk pack of the host. The archive is either an IBM 3850, CDC 38500, or equivalent.

2.2 HARDWARE

The NSS itself is embedded in this system, and its block diagram is shown in Figure 2-1 with transfer rates shown in Figure 2-2. It consists of:

- 512 processors. Each processor is individually capable of 3.6 million floating point numerical results per second, on 48-bit words, and is capable of a significant amount of overlap between index calculations, floating point operations, and memory cycles. Each processor contains units for arithmetic, indexing and instruction decoding, the processing element (PE), a 16K word data memory (PEM, or processing element memory), and an 8K word program memory (PEPM)

- 521 extended memory (EM) modules. The extended memory contains 34,144,256 words of memory in 65,536-bit dynamic RAM semiconductor memory chips.

- A transposition network (TN). Links the processors with the extended memory.

- A control unit (CU). Controls the transposition network and synchronizes the actions of the PE's, and loads and controls the PE's under certain circumstances. The control unit contains its own memory (CUM) of 32K words.

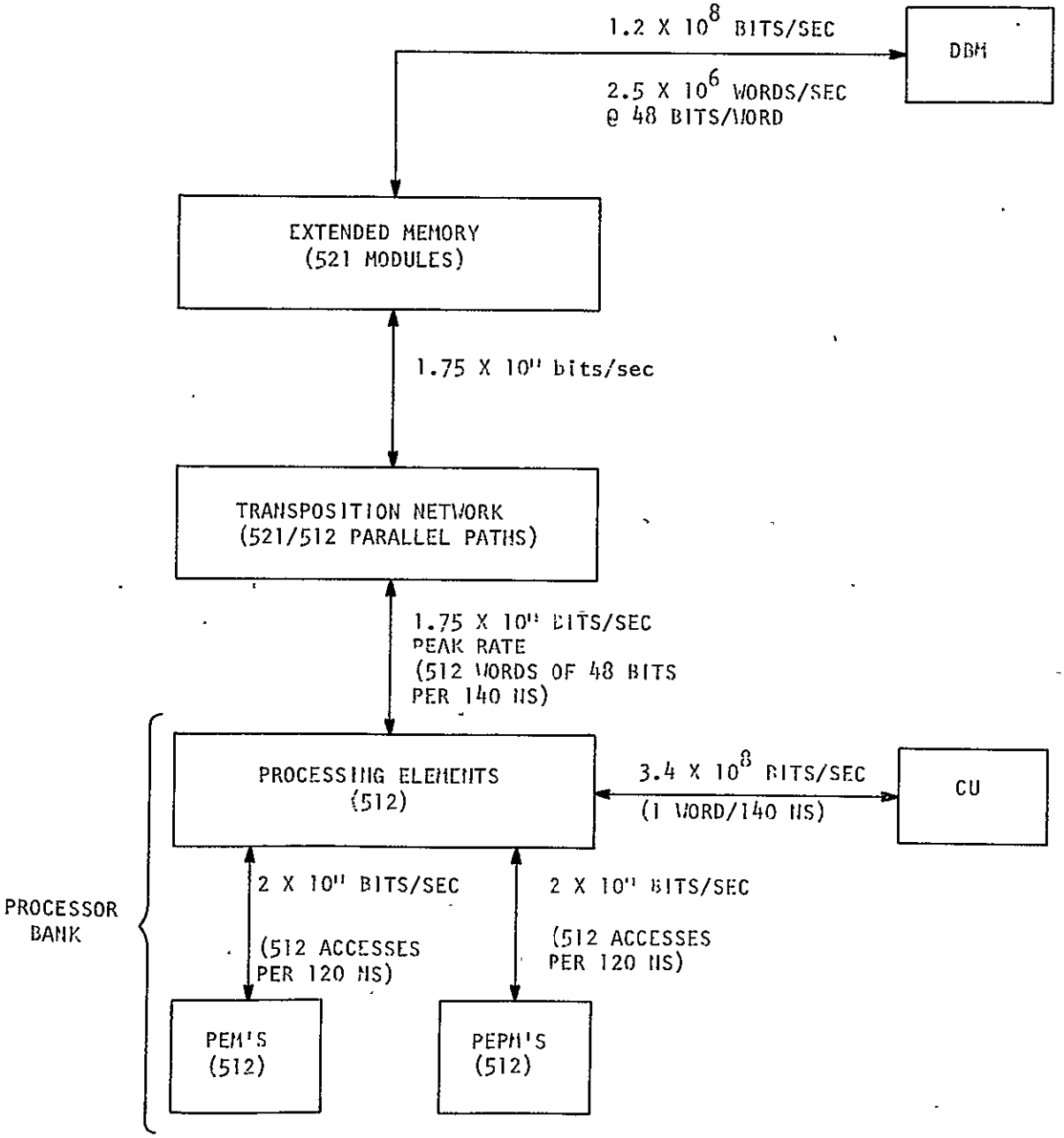


Figure 2-2. Transfer Rates

- A data base memory (DBM). The DBM is a staging area where the next task is assembled, and results are sent. It has over 10^8 words. A transfer rate of 140, 000, 000 bits per second (2, 500, 000 words per second), is maintained between DBM and EM during these transfers.

- A diagnostic controller (DC), is supplied, with an interface with the host, so that maintenance and verification of trustworthiness can be enforced by programs resident in the host.

This choice is the result of a four-way tradeoff among quantity of hardware, difficulty of design, throughput, and flexibility. The exhibited throughput figure of 1.7 billion floating operations per second (bfops) is comfortably above 1.0 billion required, and is conservatively figured on the basis of a mix of instructions. If well-ordered multiply and add operations are used we could observe 2.33 bfops.

The compiler compiles two instruction streams which are to be executed concurrently. The first stream is for the control unit; the second is executed by all the processors in concert. At specific instructions, mostly fetches from EM and stores to EM, the two streams come together, and the control unit and all enabled processors execute the instruction in synchronism across the entire array. In the stretches of code between such synchronizations, the processors are free to execute independently, each from its own copy of the processor program in PEPM, including data-dependent branches independently of the other processors or the CU.

2.3 SEQUENCE OF OPERATIONS

There seem to be two classes of users, in general, algorithm developers and production users. Algorithm developers and experimenters do much more compiling for the NSS. They may make occasional single very long runs. They will do more data reduction on their data, when they get it.

The production user will probably use most of the machine's time. He will use NSS programs that are on file. Interactively from his terminal, he sets up an experiment. This includes body geometry and grid geometry.

Having determined a suitable set of experiments, the user then describes these experiments to the NSS scheduler (or to CFD's NASF operator who, at some appropriate time, will describe them to the NSS scheduler.).

If the proper program is not in place in program memory and CU memory, it is first loaded.

Two paths for loading programs into the NSS exist. The B 7800 can cause program to be loaded into the CUM by means of the DC. Alternatively, code executed by the CU can transfer code from DBM to CUM via the EM. It is planned to use the DC path to load a bootstrap into the CU, after which the CU will load the rest of the program from DBM. The processors' code file can either be broadcast from the CUM, or broadcast from the EM. Broadcasting from the CUM is faster, since the transposition network setting must be incremented between each word when broadcasting a sequence of words in parallel to all processors from EM.

Data, which has been staged by the host in DBM, is transferred to EM by an initialization phase of the program. After that, program is self-contained within EM, CU, and processors, except for emitting results to DBM from time to time. Within the CUM is a table (prepared by the host) allocating whatever spaces in DBM are needed by the program for such output.

Overlays into EM, using DBM as backup, are possible. Since EM is large enough to contain the largest presently envisioned programs, it is not intended to provide such an overlay facility with the first delivery, but it is proposed for later extension.

During the course of the running of the program, data from Extended Memory is loaded into the PEM's via the Transposition Network. The transposition network passes two-dimensional subarrays and one-dimensional vectors from the extended memory with full parallelism. Control of extended memory modules and of transposition network is emitted from the control unit. Extended memory module addresses are computed in the processors.

When faults or error are detected during the running of the program, two courses of action are possible. For those cases which are likely to be transients, the program is terminated, and restarted from the last restart point, if any. As later analysis will indicate, only programs with running times of longer than 15 minutes or so should have restart points. For those cases where hard error is suspected, or in the case of a restarted program with a repetition of failure type, the program is terminated, and the failed processor removed from the system, and a good processor inserted instead. Manual replacement is rapid. However, automatic on-line replacement appears not to be difficult either, and will be considered for inclusion in the design. Software will make a determination as to which of these two actions is to be taken, depending on the nature of the detected fault. The self-contained nature of the processors, and the fact that only four module types (processor, EM module, DBM module, TN logic module), make up over 90 percent of all components, make for a very simple remove-and-replace repair philosophy. After replacement, diagnostics are run.

2.4 SOFTWARE

The operating system is resident on the B 7800 except for a small part that resides on the NSS and handles loading and interrupts. A listing of operating system requirements is found in Chapter 4.

The language is an extended FORTRAN. The extensions are suggested by the nature of the program, and by the ways in which the yardstick programs fit onto the NSS. An essential feature of the architecture is the ability to execute in parallel on two dimensional elements of a three-dimensional set of data. Therefore, a feature of the language is statements that cause parallelism on two indices simultaneously. For example, "DOPARALLEL J=1, 100 DOPARALLEL K=1, 100" are two successive statements that tell the compiler that all 10, 000 J, K pairs correspond to computations that can be done in parallel. The compiler then takes this 10, 000-wide parallelism and cuts it into twenty 512-wide pieces. Neither the iteration of 20 nor the width of 512 is visible to the programmer.

2.5 FAULT TOLERANCE, TRUSTWORTHINESS

Reliability calculations (very approximate, at this writing) indicate that the mean-time-between-failure (MTBF) for hard failures is high enough that the NSS needs only error detection, followed by repair, as a defense against hard failures.

For transient failures, exactly the opposite is true. In Appendix C, analysis of how infrequently can failures be tolerated, indicates that error correction and error detection will have to be widespread throughout the NSS. An apparently successful run with wrong results is far worse than an aborted run.

The prime defense against transient errors is proper design. However, the use of LSI means that the designer can no longer specify with 100 percent assurance the absolute maximum limits of values of individual components, since the equivalent resistors, transistors, etc., are not accessible to measurement. Thus, design can only guarantee some very low, but never exactly zero, transient error rate.

From analysis and conclusions of Appendix C the following design is abstracted.

- Every memory has both error detection and error correction. For PEM, PEPM, and CUM parity plus retry may save significantly in components over a Hamming-plus-parity (SECDED). However, if retry is ineffective in correcting errors, SECDED must be used.

  For EM SECDED is selected. If the error rates of the EM memory chips are much higher than estimated, a double error correction, triple error detection code must be used at the expense of more check bits.

  For DBM, the characteristics of CCD chips will force us to periodically read the contents of DBM and correct all the correctible errors, before additional errors make them uncorrectible. Whether SECDED or a more powerful error correction code is needed is to be determined. A periodic correction cycle of seven minutes is expected.

- For processors and CU, a repertoire of consistency checks is listed in Appendix C. These are in the baseline system design.

- Software checks (conservation of energy, conservation of mass, etc.) are recommended.

- Residue checking on arithmetic operations will check much, but by no means all, of the arithmetic logic in the processors. It also clearly interferes with processor throughput. A tradeoff analysis between these two effects is planned for Phase II of the NASF design, so as to achieve maximum usefulness of the NASF.

- Data is initialized to "invalid".

- Data transfers between processors, CU, and EM are covered by the same SECDED code used in the EM.

CHAPTER 3

HARDWARE

3.1 INTRODUCTION

The characteristics of the hardware elements of the NSS are tabulated in Table 3-1, and described in more detail in the sections below. Hardware is here defined as both the functional characteristics of the machine, both block diagrams and logic design, and the physical, or nuts-and-bolts aspects. Physical issues are critical to the success of the NSS design, and are therefore discussed in their own section at the end of this Chapter.

3.2 PROCESSOR

Each processor has three component parts. These are the logic portion, or Processing Element (PE), the data memory or Processing Element Memory (PEM), and program memory (PEPM). All three parts are expected to fit within 200 integrated circuit chips, and therefore will fit on one module no larger than a single large printed circuit board.

A key element in packaging this combination of elements on a single printed circuit board is the use of Large Scale Integration (LSI) circuits. It is estimated that there will be the equivalent of 13,000 gates in the processing element. To package all these gates in an estimated 100 packages or less will require the use of LSI at two hundred to a thousand gates per package, possibly including customized

gate arrays. Only a few Medium Scale Integration (MSI) packages, and practically no Small Scale Integration (SSI) packages, can be used. Commercially available LSI will be used as far as possible, but standard gate arrays with custom metallization may be needed. Appendix E describes a design using today's ECL 100K circuits with less than double this projected parts count for the PE.

Memory in the processor will use 16K-bit chips. Therefore, the PEM requires 49 memory chips; the PEPM requires 25 memory chips. Some controls are also required.

Table 3-1. Characteristics of Hardware Elements

Processor

- Number: 512
- 48 bit data word
- Multiply time, 360 ns; add time, 240 ns; add product 440 ns (multiply and add); for normalized, rounded floating point operations.
- One large p/c board for the processor includes PE, PEM, and PEPM.
- Approximately 100 integrated circuits in PE.
- Overlapped instruction fetching and decoding, partially overlapped index arithmetic.

Processing Element Memory (PEM) and Processing Element Program Memory (PEPM)

- Number: 512 of each on same boards as PE's.
- Memory cells: 16K-bit integrated circuit chips, (49 chips and 25 chips, respectively).
- Cycle time: 120 ns.
- Error control: parity checking with retry upon failure, error halt on second failure.

Extended Memory (EM) modules

- Number: 521 modules
- Memory cells: 64K-bit RAM chips, 56 chips per module, 65536 words per module.
- Cycle time: 260 ns single word, interlaces for 140 ns/word for block transfers.
- Capacity: 65, 536 words per module, 34×10^6 words total.

Table 3-1. (Cont'd)

- Error control: Hamming plus parity for single error correction, and double error detection. Error detectors are in PE, DBM controller. Error corrections are logged.
- Transfer method, 7 bytes per word, one byte per 20 ns, to PEM, word-parallel to DBM bus.
- Auto-incrementing address register for streaming fetches.

Transposition Network (TN)

- Block transfer rates, one word per 140 ns.
- Path width, 8 bits per path, 521 paths to EM, 512 paths to processors.
- Control setup time, 80 ns (measured from start of controlling instruction in CU).
- Delay from EM to PEM, in streaming mode, 80 ns.

Control Unit (CU)

- CU memory: same technology as PEM, but 32,768 words.
- Average instruction execution time: 350 ns (estimated).
- Complexity: 30,000 gates
- Synchronization time: 180 ns.

Data Base Memory (DBM)

- Implementation, 256K-bit CCD chips.
- Capacity, 6×10^9 bits.
- Transfer rates, 140×10^6 bits/sec to/from EM. Unloads into disk-pack file system of host processor on other side.
- Controller accepts input from CU for DBM-EM transfers, accepts input from host I/O channel for DBM-host transfers, and resolves conflicts between the two. Design of the controller is dependent on choice of host processor (whether the host buffer DEM-disk pack transfers, or whether buffering resides in the controller).

Diagnostic Controller

- Accepts single instructions from diagnostic program running on host.
- Overrides any CU action.
- Manipulates PE's and EM by causing CU to execute single instructions.

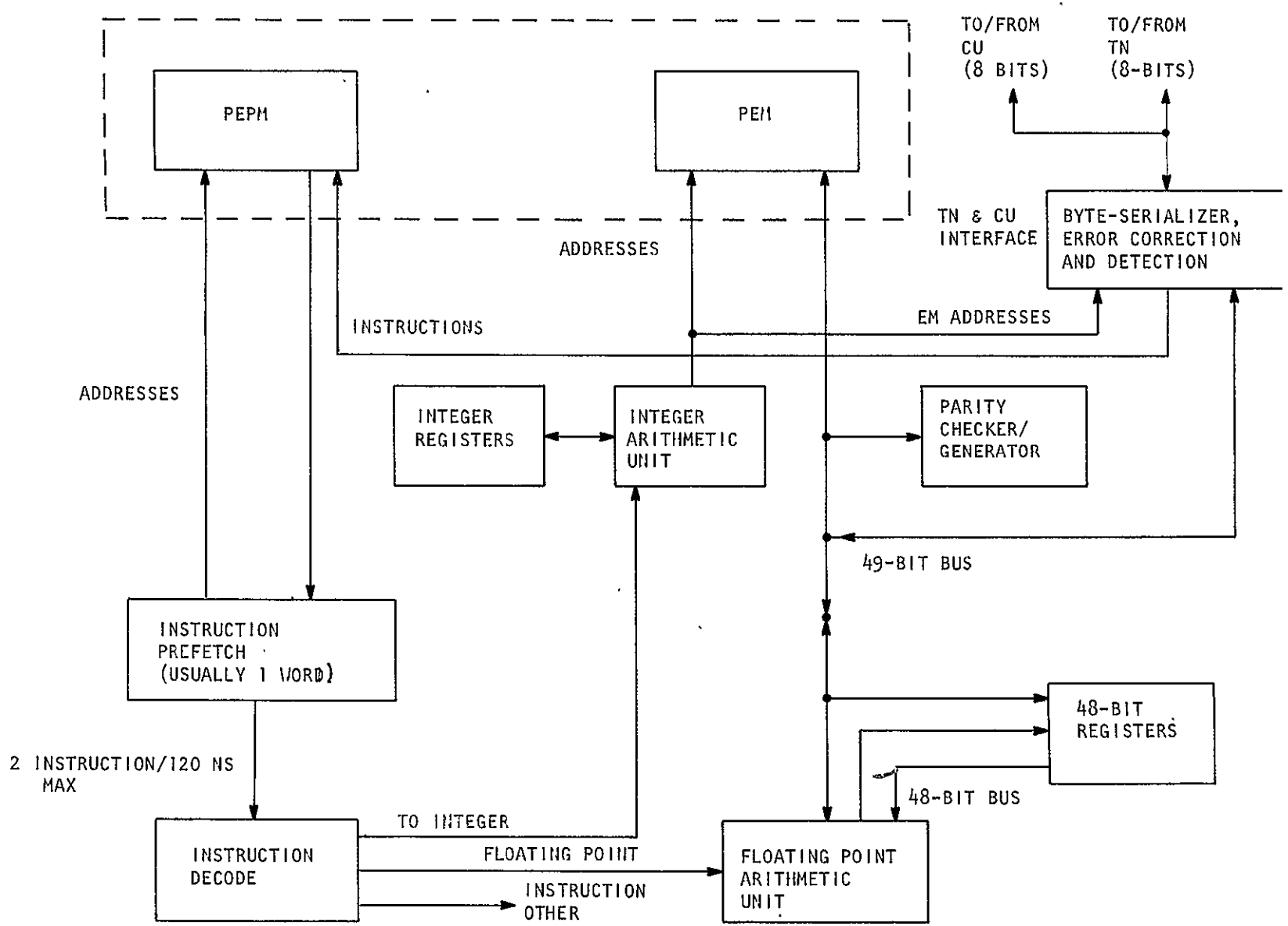


Figure 3-1. Internal Block Diagram of PE

3.2.1 Processing Element (PE)

The main features of the PE design are:

- Optimum arithmetic speed.
- Local registers to reduce stores and fetches to PEM.
- Separate index arithmetic section and index registers for overlappability of address calculations.
- Simple CU interface, only a few signals.
- Independent, overlappable byte serializer and error checker for the Transposition Network interface.
- Barrel switch for fast alignment of addends and normalization of results.
- Error correction and detection at several points.
- Independent, overlappable instruction fetching.

Figure 3-1 shows the general arrangement of the sections within the processing element. Instruction fetching and decoding form one partially independent section. The 16-bit integer registers and a 16-bit adder form a second independent unit. The floating point unit is the most complicated of all the blocks shown in Figure 3-1. In it, there is a separate adder for exponents, and multiplication hardware. Multiplication is disucssed in a section of Appendix D. A 39-bit (plus guard digits) adder and a barrel switch needed for alignment are included. Associated with the floating point arithmetic units is a bank of eight 48-bit registers. The path to processing element memory carries addresses from the integer arithmetic unit while a bidirectional 49-bit-wide data bath is connected to the register file, to the parity checker-generator, and to the transposition network interface. The TN interface is also used for converting words in and out of byte-serial form for transfer to and from the CU.

The processing speed, and therefore the number of processors, arises from the speed of economic memory.

3.2.2 Instruction Handling

In designing the PE, a goal is to get as much overlap of operations as possible while keeping the design of the controls relatively simple. As always in specifying solutions to engineering problems, optimization results from tradeoffs. Figure 3-2 shows the chosen mechanism for achieving a reasonable amount of overlapped operation with relatively simple controls.

One, two, or three instructions are in various stages of execution, in the integer arithmetic controls, the floating arithmetic controls, or in the PEM controls. One word's worth, either one full word instruction or two half-word instructions are in the instruction staging register.

When any of the three semi-independent execution stations finishes, and if the next instruction is for that station, and if the instructions at the other two stations will not interfere with further action at the idle station, then the next instruction is passed to the free station, or in the case of the PEM fetch controls, the address field of an address-bearing instruction is passed.

When the first half-word instruction is emptied out of the Instruction Staging Register, the second half-word becomes the "next" instruction.

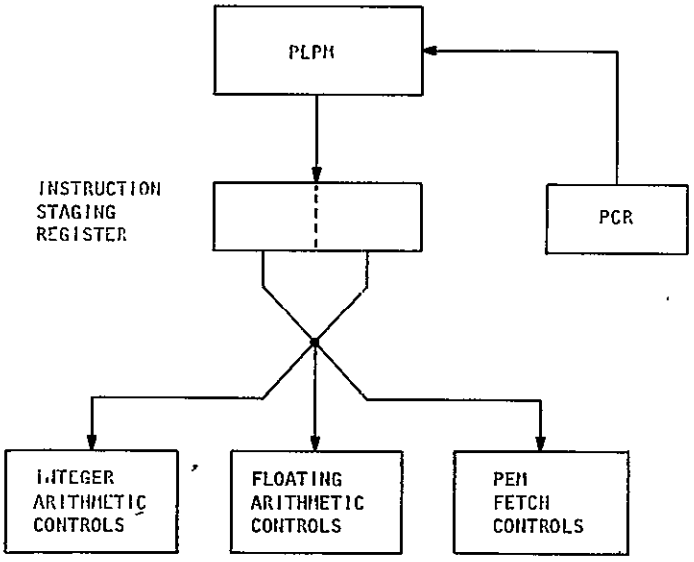


Figure 3-2. Instruction Fetching Machines

When all of the Instruction Staging Register contents have been passed to the relevant execution station, a new word is loaded from memory. Whenever a word is loaded into the instruction staging register, the program counter (PCR) is incremented, and the next address in PEPM is caused to read. Thus, PEPM read access time is usually overlapped with instruction execution. To keep the instruction handling simple, the binary code used for each operator adheres to certain rules:

1. The first bit of each instruction is "1" for full-word and "0" for half-word.

2. The second bit of each instruction is "1" if the address field is an 18-bit field and is used for fetching. This bit, plus the "00" or "01" at the beginning of the address field indicates that a PEM fetch is to be executed. PEM access time can thus be overlapped before the rest of the instruction is decoded.

3. The next bits of the instruction are "00" for index arithmetic operations, "10" or "11" for floating-point operations, and "01" for other cases.

The timing information in the software chapter is based on particular designs for the logic, which is discussed at greater length in Appendix D on logic design. In particular, a multiplication algorithm for the baseline system is disclosed.

3.2.3 Processing Element Memory (PEM)

The processing element memory is 48 bits per word plus parity. A fault detected by the parity checker will result in the refetching of the fetched word, and the disabling of any results in the next PE major clock. Each faulty parity thus costs 120 extra ns, but only in the PE in which the fault is found.

Since there are 16, 384 bits per chip, and 16, 384 words in the memory, the obvious organization, of one chip per output bit, is adopted. Chip address controls are connected directly to the memory address register, which is designed with sufficient drive capability to drive all chips in parallel. Two of the address bits are used to select one of the four output bits of each chip.

Logic surrounding the individual chip is shown in Figure 3-3.

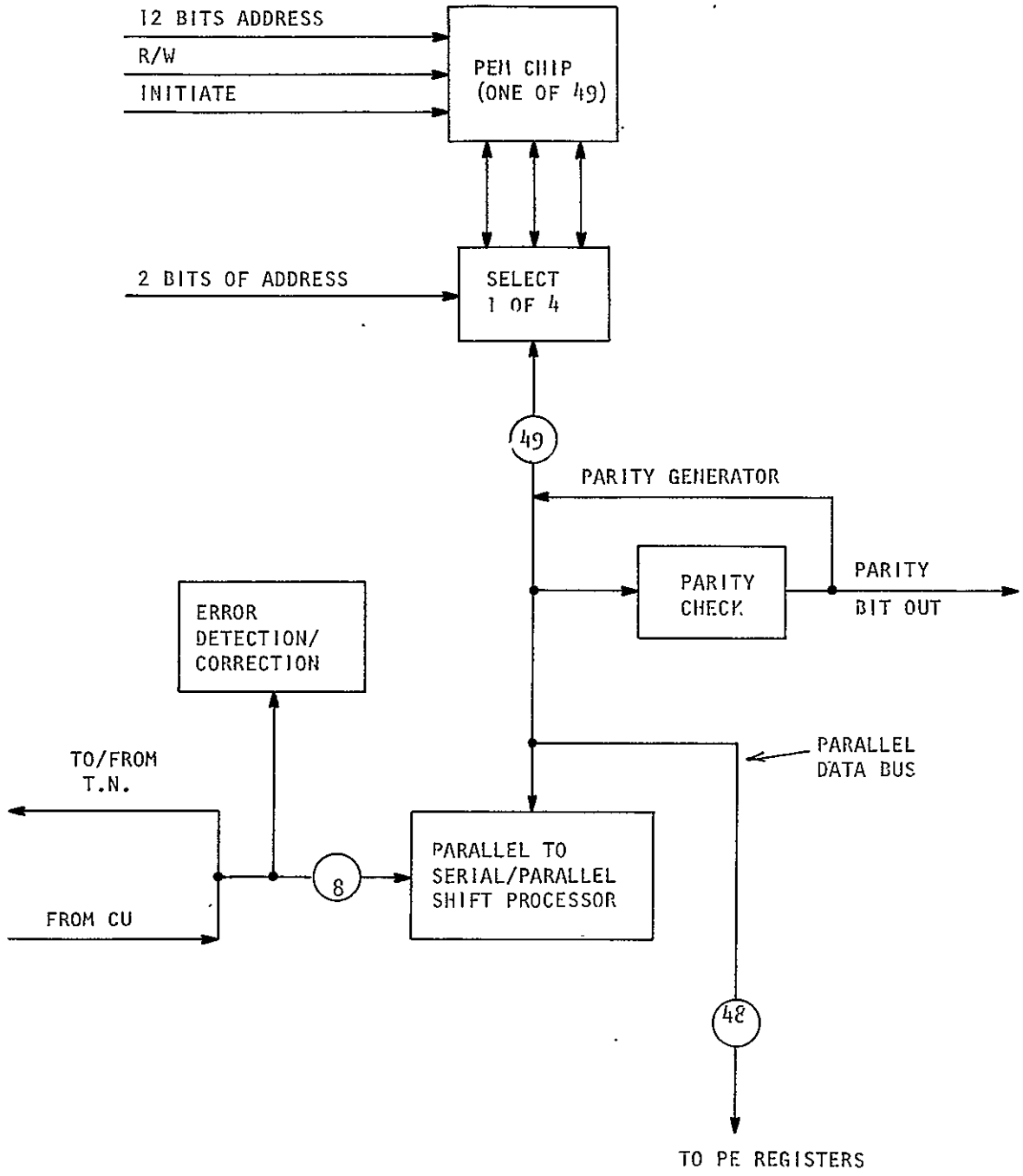


Figure 3-3. PEM Logic

Cycle time is 120 ns, giving 8,300,000 words per second per PE. Since there are 512 PE's and 49 bits per word, the transfer rate is 2.0×10^{11} bits/second for the entire parallel channel.

Although the normal word length is 48 bits, some data is packed in two 24-bit words per memory word, giving an effective size just slightly larger than the specified 16,384 words.

3.2.4 Processing Element Program Memory (PEPM)

The processing element program memory is the same as the PEM except for being half as many words. Program addresses within PEPM are fixed at linking time, not dynamically assigned at overlay time, so that absolute addresses can be used; that is, the program counter is used directly as the PEPM address register.

A full word is fetched at each cycle. Some instructions are full word, some are half word. The word length is the same, 48 bits, as in PEM.

Loading PEPM from the CU can be done at the rate of 140 ns/word, from the CU memory. Therefore, it is not inefficient of time to overlay program. The full 8192 words of every PEPM are loaded in only 1.2 ms.

3.2.5 Processor Interface

The processor board has a card-edge connector with about 90 pins. Thus, there are relatively few connector junctions in this machine, considering the amount of circuitry involved.

The pins are assigned as follows:

8 signals from the CU for broadcast variables

8 signals to the CU for data transmission

9 signals to the transposition network. (8 data plus strobe)

9 signals from the transposition network

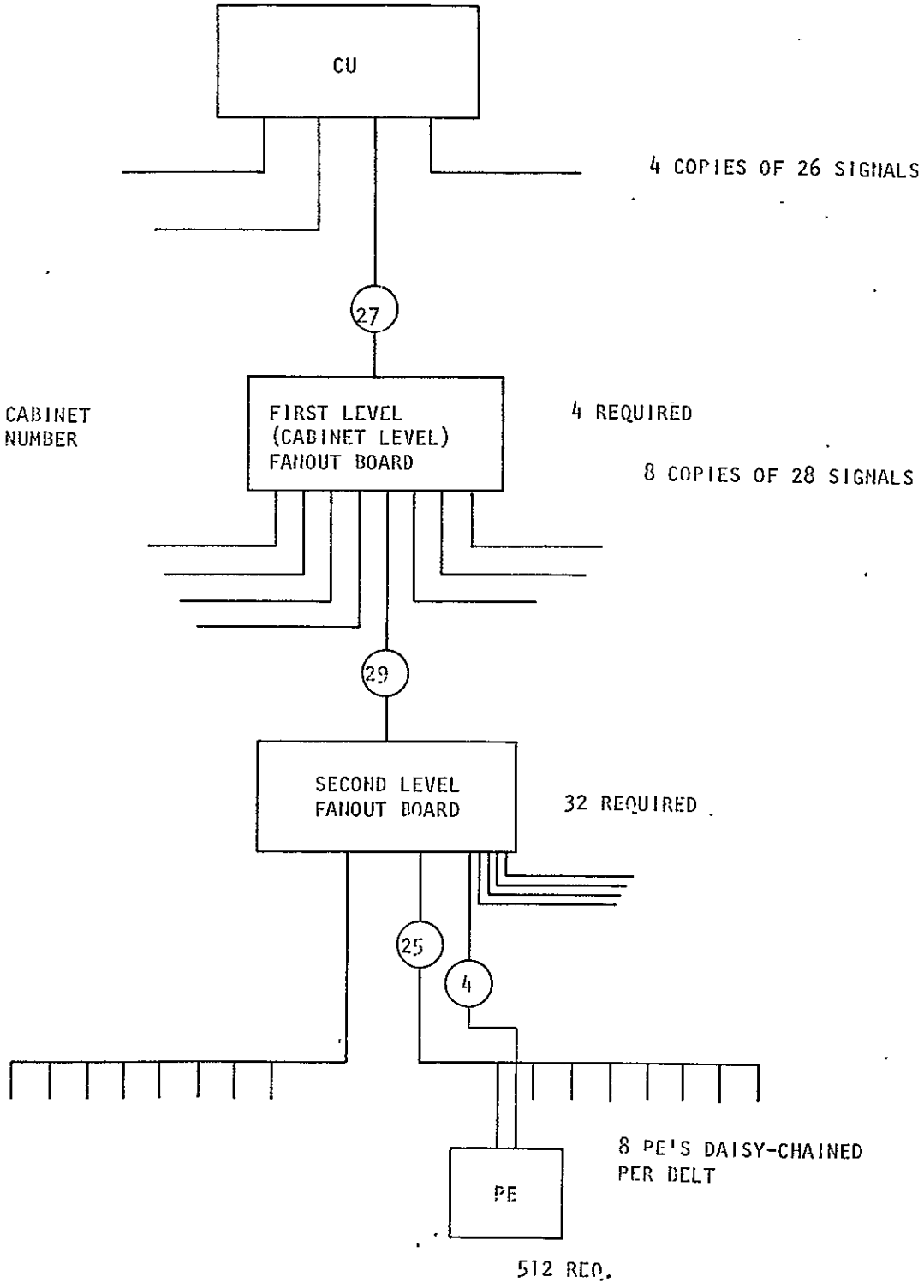


Figure 3-4. Fanout Tree

6 one-bit lines "on", "running", "interrupt CU", "fault", "I got here", "go".

4 bits of control code from the CU. Codes include "load program counter", "halt", "clear memory address register", and others.

9 lines given processor number. Some of these lines come from the cabinet-level fan-out board so that all cabinets are wired identically.

25 (approximately) power and ground connections. Multiplicity of ground connections is needed for the mechanical (and electrical) termination of 8-signal belts.

10 spares.

Between the CU and the processors there is a set of fanout boards. These boards provide the fanout function so that signals being broadcast need not have 512 wires each coming from the CU. These boards also contain the necessary logic functions of signals being sent to the CU. For example, every "on" PE is to have its "running" bit ORed together so that CU can tell if any PE is running. This requires the 512-input OR of "on" AND "running". The cabinet level fanout board contains the cabinet part of PE number, the least significant end of the number being wired into the backplane.

The wiring consists of one 18-signal belt to the transposition network and one 24-signal belt to the last-level fanout board, which is daisy-chained to a number (typically 8) of other PE's. Some of the PE signals ("on" and "running", etc.) are logically combined at the fanout board and cannot be daisy-chained.

The least significant end of PE own number is wired at the backplane connector to "one" or "zero", these being two voltages supplied from the PU board.

Figure 3-4 shows the fanout tree for the connection between CU and PE's. The CU has 4 groups of 27 signals. Most of the matching signals in each group are replications of the same signal. The CU uses 108 signal pins to communicate with the array of PE's.

In each cabinet of 128 PE's there is a cabinet-level fanout board. This board has 27 connections to the CU, two signals for the "cabinet number" jumpers wired into the backplane, and eight copies of a 29-signal interface to the "row" fanout board, for a total of 241 connections.

The "row" or second level fanout board has a 29-signal interface with the first level board, two 24-signal belts that daisy-chain to eight PE's each, and 16 copies of a 4-signal interface to those same 16 PE's. The result is a signal count of 141 before ground, power, etc. are added.

3.3 TRANSPOSITION NETWORK

The transposition network (TN) accepts vectors whose elements are not in contiguous addresses, and passes them in parallel to the processors as though they were contiguous. For fairly general types of vectors, there are no memory conflicts. The vector elements are emitted in parallel by the EM modules. The TN rearranges their sequence on the way to the processors. Design of the transposition network has to face several issues:

1. What patterns of scrambled order of data elements, coming from the EM modules, must be unscrambled in getting to the processors?
2. What logic design gives the best tradeoff among parts count, flexibility of connection patterns, and ease of control?
3. What is the tradeoff between parts count and bandwidth?

These issues are discussed in turn.

3.3.1 TN Requirements

Addresses in extended memory are continuous, in terms of address space. For a single address A, the EM module number M is given by

$$M = A \bmod 521$$

For a set of up to 521 addresses A_i ($0 \leq i < 521$) where $A_i = A_0 + p*i$, we have module numbers $M_i = (A_0 \bmod 521 + (p*i)) \bmod 521$. Swanson (see reference) calls this a p-ordered vector, starting at M_0. In the terminology used in the Burroughs Scientific Processor (BSP) project, p is the "skip distance".

The requirement for the transposition network is simply this. A set of memory addresses, each separated by a constant amount p, is found in different memory modules. The transposition network takes the first member to the first PE,

the second member of the set to the second PE, and so on. (See Figure 3-5). To allow the maximum number of options, the number of memory modules must be a prime number. Fortunately, the number of memory modules is completely invisible to the programmer, who works with an address space described by successive numerical addresses, very conventionally.

In the section on data allocation, it is shown that more complex two-dimensional arrays can also be fetched in parallel using this same unscrambling of p-ordered vectors.

In addition to unscrambling these p-ordered vectors, the transposition network must also transmit in the opposite direction, for passing addresses from processors to EM modules, and for storing. In addition, there is a broadcast mode, in which a single word from a single EM module is transmitted to all processors.

3.3.2 Choice of the Transposition Network

Several possible transposition networks were considered. These include:

1. The selected transposition network, described below, and on wich which a patent application is now pending. It has low parts count ($2N \log_2(N)$), short delay, and is easy to control.

2. The Benes network. Parts count and delay are as good as the selected network. Control is complex, requiring over 2000 bits for each desired permutation, of which there are over 521^2. These control patterns take either a very long time to compute, or require a very large ROM

3. A version of routing, as in ILLIAC IV. This scheme is rejected for delay; up to 16 register-to-register transfers are needed to rearrange a single 256-long vector of variables, or for parts count, since the added delay can be partially overcome by additional hardware.

4. A scheme, wherein each processor has its own EM module, and neighbor-to-neighbor connections allow one processor access to the data in other EM modules. This is similar to Sutherland's scheme as described in Scientific American for September 1977. With the data accessing patterns of the benchmark programs, either a) the percentage of idle time among processors is very high, or b) the data allocation options are very restrictive, and data allocation is very difficult to figure out.

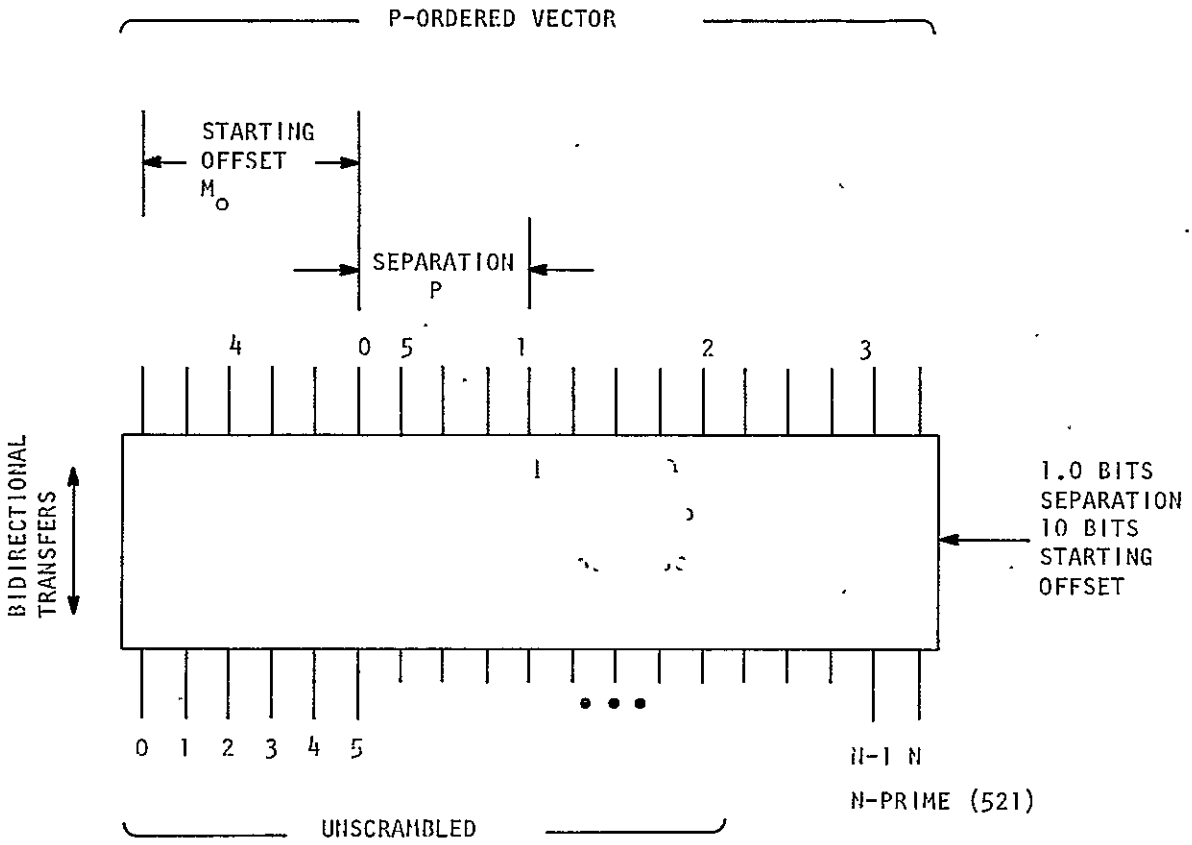


Figure 3-5. Transposition Network

5. A full crossbar between EM modules and processors. This has excessive parts count, on the order of N^2.

Table 3-2 summarizes these comparisons. The alternate schemes are discussed in Appendix B.

The transposition network has the following properties:

- Transfers between a p-ordered connection of EM modules and the direct, 1-ordered, PE modules.
- Receives p and M_0 (10 bits each) from the CU for control.
- Has delays given by the approximately 50 feet of wire between EM and PE, plus the delay through 10 gates.
- Broadcasts to all processors from a single EM module are also performed.

As becomes clear below, control of the TN requires a shift distance of 10 bits, a 10 bit control setting for the skip distance p which is read from a 520-word by 10 bit ROM, and about three bits of control to specify the action: unscrambling the p-ordered vector, broadcast, or any other combination finally specified.

3.3.3 Design of the Transposition Network

The equations relating EM module number to i, are repeated as follows:

$$M_i \doteq (s + p \cdot i) \text{ modulo } 521$$

where M_i is the physical location, or position, of the ith element of the vector being fetched. We wish to transpose this ordering in such a way that the ith element falls into position number i.

Consider the solution of the equation

$$i = (M_i - s)/p$$

Table 3-2. Comparison of Transposition Networks

Feature	Network				
	Baseline	Benes	Routing	Nearest Neighbor	Crossbar
No. of Components	$2N \log_2 N$	$2N \log_2 N$	N Factor	$7 \times N$	N^2*
Delay (clocks per transfer)	1	1	Many*	1	1
Control Complexity	Simplest	Very Complex*	Simple	Simple	Simple
Programming Restrictions	Essentially None	None	Essentially None	Severe*	None
No. of EM Modules	N = Prime	N = Prime	$N = 2^{2n}$ (Preferred)	$N < B < A$ All rel. Prime $N = AB$	Any
No. of Processors	$P \leq N$	$P \leq N$	$P = N$	$P = N$	No Constraint

*Reason for Rejecting

for the moment forgetting that the arithmetic is modulo 521, a fact which we shall reintroduce. We can solve this equation as follows:*

$$i = b^{(\log_b(M_{i-s}) - \log_b(p))}$$

where b is the base of some logarithm, as soon as we can define a suitable logarithm.

To continue this explication, let us simplify the example to 11 EM modules, instead of 521, and take 2 as the base of the logarithms, according to Table 3-3, which is in arithmetic modulo 11.

Table 3-3. Powers of 2 in Arithmetic Modulo 11

p	2^m	p	$m = \log_2 p$
$1 = 2^0 = 2^{10}$		1	0
$2 = 2^1$		2	1
$4 = 2^2$		3	8
$8 = 2^3$		4	2
$5 = 2^4$		5	4
$10 = 2^5$		6	9
$9 = 2^6$		7	7
$7 = 2^7$		8	3
$3 = 2^8$		9	6
$6 = 2^9$		10	5
$1 = 2^{10}$			

The exponents of 2, in Table 3-3, can be used as the logarithms of the numbers on the left hand column, in performing modulo 11 arithmetic. We solve our equations as follows, and translate it into hardware as shown in Figure 3-6.

*Essentially equation 148 on page 110 of Shanks book on number Theory (Reference 26).

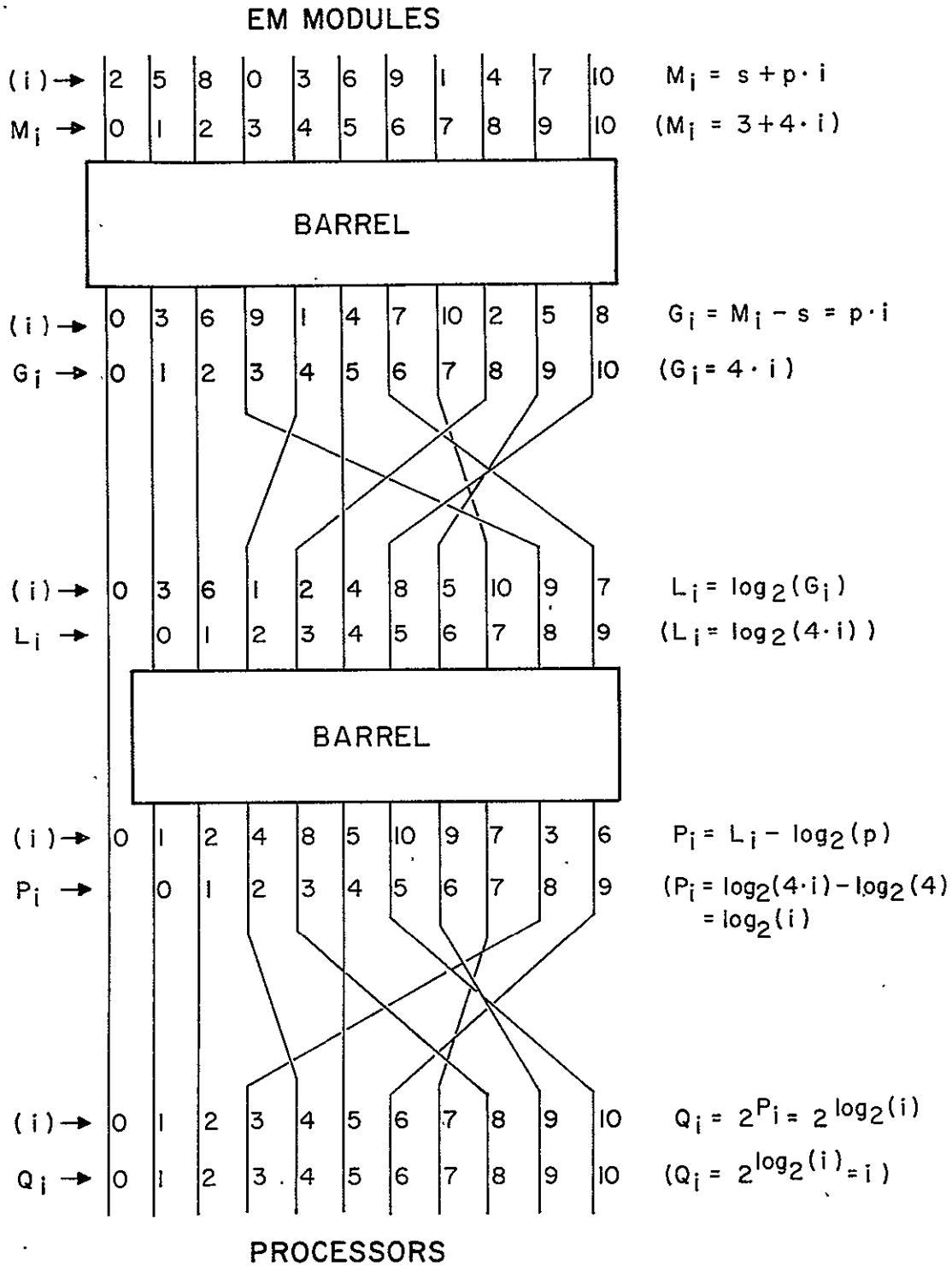


Figure 3-6. Transposition Network Functioning for N=11

1. At the top of the figure, the module number M_i is given by $s+p \cdot i$. A numerical example accompanies the figure, with s=3 and p=4.

2. After passage through a conventional barrel switch, the positions are given by $G_i = p \cdot i$. That is, a shift left of s subtracts s from each signal's position. The barrel is 11 wide and shifts end around.

3. A wiring scramble, a permanent part of the backplane, takes every position G_i and converts it into a new position $L_i = \log_2(G_i)$ where "log" is defined by Table 3-3. Note that $2^{10} = 2^0$, the barrel is 10 wide and end around therefore. The 0 position is not involved, since the barrel already put i=0 in the 0 position, and the $\log_2$ function is not defined for $\log_2(0)$.

4. A second barrel now subtracts $\log_2(p)$ from the position L_i. For example, p=4 and $\log_2(4) = 2$ by Table 3-3, so the barrel shifts left by 2. Table 3-3 is stored in a ROM in the CU.

5. A second wiring scramble, the inverse of the first, finishes the transformation.

Each step in the solving of the equation takes the physical location of the 11 signals and rearranges them to a new set of physical locations; the successive physical locations being the M_i, G_i, L_k, P_i, and Q_i respectively. Down the right side of Figure 3-6 is the algebraic representation of the separate steps; down the left side is indicated what the numbers on each line mean.

The extrapolation from 11 EM modules to 521 EM modules is shown in Appendix B in Table B-2. Here, in modulo 521 arithmetic we find that powers of 2 do not cover the 520 values, but we find that powers of 3 will. Everything else remains the same. The upper barrel has width 521. The wiring scramble is described by Table B-2. The lower barrel has width 520.

An additional detail in the design of the TN is the provision for broadcasting. Module number s is moved to position 0 on the output of the upper barrel. An extra path is inserted from position 0 at the output of the upper barrel, to position 0 of the input to the lower barrel. All other outputs of the upper barrel are forced to zero. All control signals, for all shift distances, are enabled in the lower barrel, so the lower barrel's outputs, all of them, are the OR of all the lower barrel's inputs. Only one input is not zero, therefore the output of the lower barrel is equal to that one input.

Some simplification of controls can be achieved if the barrel chips latch their outputs on command, but such latching is not a required feature. The latching control is used both a) to latch data as it passed through for retiming purposes, and b) to disable the level by latching onto previously established zero inputs. The Fairchild 100158 shift matrix is capable of implementing the transposition network, but somewhat inefficiently, since the individual chips shift end off, and all gates must be duplicated in order to achieve end-around shifting.

The path width of 8 bits, and a per-byte clock of 20 ns, means that 140 ns is required to transfer an entire word of 55 bits. Seven bytes are 56 bits, leaving one spare bit location in the transferred word.

The above description of the transposition network differs considerably from the description in the patent application*, which involves an extension of the ideas in Swanson's article. The two descriptions are of the same connections between individual gates, and therefore of the same transposition network. This alternate description is included in Appendix B.

3.3.4 Required Bandwidth

The transposition network described has an 8-bit wide path, so that seven successive transfers are required per word. This is designed to match the postulated EM speed (260 ns cycle time memory chips, with an interlace of two submodules per EM module). Since the TN is a much smaller amount of hardware, and less cost, than the sum of the EM modules, it makes no sense to make EM performance suffer because of limitations in the TN. Therefore, the TN is designed to keep up with EM cycle time.

On access time, we note that the access time to EM, as seen from the processor, includes the EM access time (a large fraction of 260 ns) plus the round trip delay from processor to EM and back, of 100 ns. The TN design adds 120 ns to this access time because of the byte serial nature of the word returned through it. Since TN transfer times are only one or two percent of the total execution times,

*A patent application has been applied for on the transposition network. This work was done under the subject contract.

a wider TN, with reduced access times, is not an effective investment in hardware for speedup purposes.

3.4 EXTENDED MEMORY (EM)

Each extended memory module contains two submodules of 28 memory chips each, each chip contains 64K-bits, so that each EM module contains 65,536 words. Since there are 521 modules, there are 34×10^6 words total.

Each EM module is split into two halves for interlacing: Odd numbered addresses are in one submodule, even numbered addresses are in the other. Each half consists of 28 chips, with two bits of data per chip, and one half chip unused.

The EM module interfaces the transposition network on a byte-serial connection 8 lines wide, and likewise on the DBM side. On the TN side, it executes the following operations on command from the CU (these operations being portions of CU instructions):

- LOAD the address register from the TN while loading the increment register from the CU
- FETCH one word to TN from the address given, and increment the address.
- STORE one word received from TN to the address given, and increment the address.
- LOAD the word received from the TN into the data register, do not store.
- SEND the word in the data register to the TN, do not fetch.
- SEND module number (wired into the backplane) into the TN.

Error correction code generation and detection is done in the DBM controller and the PE. It does not affect the EM module at all.

On the DBM side, the EM module responds to just three commands, sent from the EM control portion of the CU:

- LOAD address register.
- FETCH one word, increment address by 1, data register is connected to the 140 megabit bus that goes to DBM.
- STORE - Accept one word from 140 megabit bus and store it. Increment the address by 1.

At one time, a shift register implementation, using 64K-bit CCD chips, for the EM was considered. No satisfactory tradeoff between block size, access time or transfer time could be found. A RAM implementation of EM is clearly superior.

3.5 CONTROL UNIT (CU)

The control unit executes a set of instructions of somewhat greater diversity than those that belong to the PE, but has no need for the complexity of floating point hardware.

The control unit includes the following functions:

1. Integer arithmetic for address claculation and loop control.
2. Synchronization with PE's; commands sent to the PE's.
3. Controller for EM-DBM transfers; resolution of conflicts with DBM-host transfers initiated by the host.
4. Interface with host I/O channel for communication with host-resident programs and for loading NSS programs.
5. Machinery for broadcasting program to the PE's.
6. Local memory.

In addition, the CU can be manipulated by the diagnostic controller for diagnostic purposes.

The CU memory will be manufactured of the same technology and will share parts types with the PEM. The CU cycle time will be 40 ns. A homogeneous CU memory allows the allocation of space to program and data in an optimum manner. The size of the CU, perhaps 30,000 or 40,000 gates, is small enough that redundancy of operation is not necessary although error detection is desirable. Several memory bounds separate the operating system, the CU's own program, program files for loading into the PE, and CU data. CUM can be loaded either by the DC or from EM.

3.5.1 Synchronization

Synchronization requires that all processors and the control unit wait until the last one of them is ready to perform some action, whereupon the last one's being ready triggers the entire set into synchronized action. Synchronization can be accomplished in two ways, depending on whether that last ready element is one of the processors, or is the control unit.

If the last ready element is expected to be the CU, we call this "type I" synch. It goes as follows. As each processor reaches the point in the program at which it waits, it raises an "I got here" flag. When the CU reaches the point in the CU program that corresponds, it checks all the "I got here" flags from enabled processors (unenabled processors don't count, so the condition tested is the 512-way AND of "I got here" OR NOT "enabled"). When all "I got here" bits are detected as being true, the CU issues a "go" signal which is received at all processors simultaneously.

If the last ready element is expected to be one of the processors, this is called "type II" synch. The CU reaches the point in the program where the cooperating process is to start, and performs some action of its own (typically, setting the TN to some unscrambling pattern), and sends an "all is ready" signal to all processors. The processor, sensing the "all is ready" signal, continues.

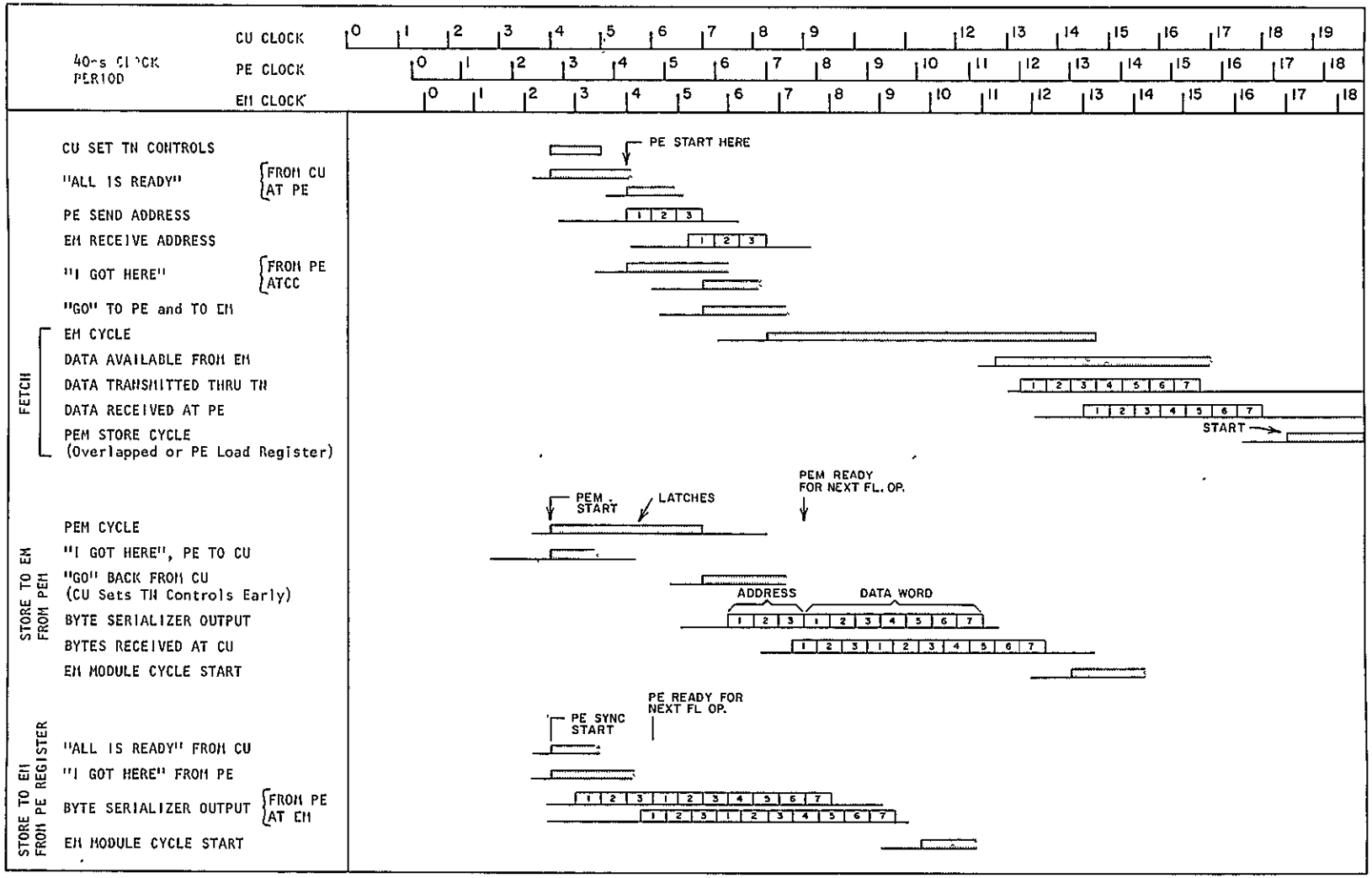


Figure 3-7. Synchronization

The EM fetches and stores use a mixture of both. The CU sets the TN controls and issues "all is ready". The PE's transmit addresses to the EM modules simulatneously with sending "I got here" to the CU. When the CU sees the AND of all the "I got here's" coming true, it knows the last address is being loaded, and simultaneously transmits a timing pulse to EM, and "go" to the PE's. For fetches, the result is that the PE's time their memory cycles in PEM in synchronism with the data being received through the TN from the EM's. For stores the relative timing at the EM is different by a constant delay. See Figure 3-7. When N=1, an option is for the PE to load a register instead of PEM.

3.6 DATA BASE MEMORY

3.6.1 Requirements

The Data Base Memory (DBM) stand between the host and the archive on one side, and the rest of NSS on the other. Its controller accepts commands from the host for transfers to and from the host, and to and from the archive if a separate I/O channel is provided directly to the archive. The controller accepts commands from the NSS's control unit for transfers between DBM and the extended memory in the NSS. The controller also contains mechanisms for error correction and error control.

Requirements are a transfer rate of about 10^8 bits per second to and from extended memory, and to match whatever other sources the host may having including file storage, on the other side. Although the memory of the host computer is available to be used as a buffer between the DBM and other elements of the host system, a minimum use of such buffer areas is a design goal.

The size of the DBM is such as to hold a reasonable number of data bases, each of which is 3×10^8 to 10^9 bits in size. Thus, the minimum size is "several" billion bits. The size selected is 6×10^9 bits.

DBM could be successfully implemented in a variety of technologies. It could be CCD, rotating magnetic storage, or magnetic bubbles. A CCD version of the DBM is preferred, for a number of reasons, and is discussed below in Chapter 3 as an element of the baseline system. Rotating magnetic storage (disk) would be

the low-risk choice if the DBM design had to start today. Disk has several disadvantages with respect to CCD. Timing cannot be adjusted to eliminate buffer requirements. Access time is longer. Scheduled maintenance is required. Size is larger. Error correction codes are more complex. Magnetic bubbles apparently will require more development before they become suitable. Extrapolation from existing bubble designs indicates that bubbles will require very wide parallelism to achieve the desired transfer rates. Bubbles also require more off-chip circuitry, and more complex control than CCD's. Disks and bubbles, as alternate DBM designs, are discussed in Appendix H.

3.6.2 Implementation

Current projections indicate that by 1980 there will be 256K-bit CCD memory chips. The advantage of using CCD chips instead of disk packs is a simplification of the controller, since timing is under the control of the controller and only very small buffers will be required.

A second advantage is that the same error correction code can be used in DBM as is used in extended memory, and check bits need not be generated on transfers between the two memories, just checked.

There is a "natural" block size for a CCD memory given by the length of the internal shift registers times the number of chips read in parallel. By setting the logical block size equal to the natural block size, a number of advantages are gained. One can arrange things so that the first bit of any block is most of the time in place, so that access time is usually zero; after shifting the block around at maximum speed, we rest for a while at the "next-bit = first-bit" location. Whenever the next access comes quickly enough, and it will during EM-DBM transfers, there is zero access time to the next block. Using natural block sizes also simplifies the addressing controls, since logic address and the CCD chip control bits are identical.

To keep the block size from being too large, the number of chips read in parallel must be kept down to some reasonable number. Seven chips in parallel matches a simple logic design for the error detection and correction logic, albeit somewhat

different from the byte-serial, 8-bit-parallel logic used in the EM, and also matches the block length, since eight 7-bit pieces will make up one word with Hamming plus parity error control. Each module of DBM reads seven chips in parallel, perhaps at an estimated 2.5-MHz rate, or 3.2 microseconds per word per module. With eight modules there is an average rate of 8/3.2 or 2.5 words per microsecond, giving a raw data rate of 55 bits per word or 140×10^6 bits per second. Of these bits, 48 bits per word, or 120×10^6 bits per second are actual data. Each module contains 3584 memory chips (512 by 7). The entire DBM contains 134, 217, 728 words of 55 bits each or over 7.3×10^9 bits total.

The entire contents ofDBM (CCD version) is continually read and rewritten at the single-module transfer rate (18 M bits/sec nominal), so that any spontaneous errors that arise during the CCD refresh process can be eliminated before they turn into double errors and are uncorrectable. The entire contents are scrubbed free of errors every seven minutes.

A CCD design for the DBM will require trouble free power. Depending on the nature of the a-c power at Ames, the system may need to ride through transients of only tens of milliseconds; easily done with filter capacitors. In addition, there may be a need to cover outages of many minutes requiring battery backup. Because of the need for scrubbing errors, one cannot reduce the power during standby as much as in some other systems, perhaps power is not reduced on standby at all. It is estimated that the DBM requires 500 amperes at 12 volts, and 400 amperes at 5 volts, plus other voltages at lower currents. This could easily represent a 12 kw or larger load on the batteries of uninterruptible power supply. The simplest arrangement, in which the batteries remain connected at all times, is shown in Figure 3-8. It has no 60-Hz transformers, thereby saving space and weight.

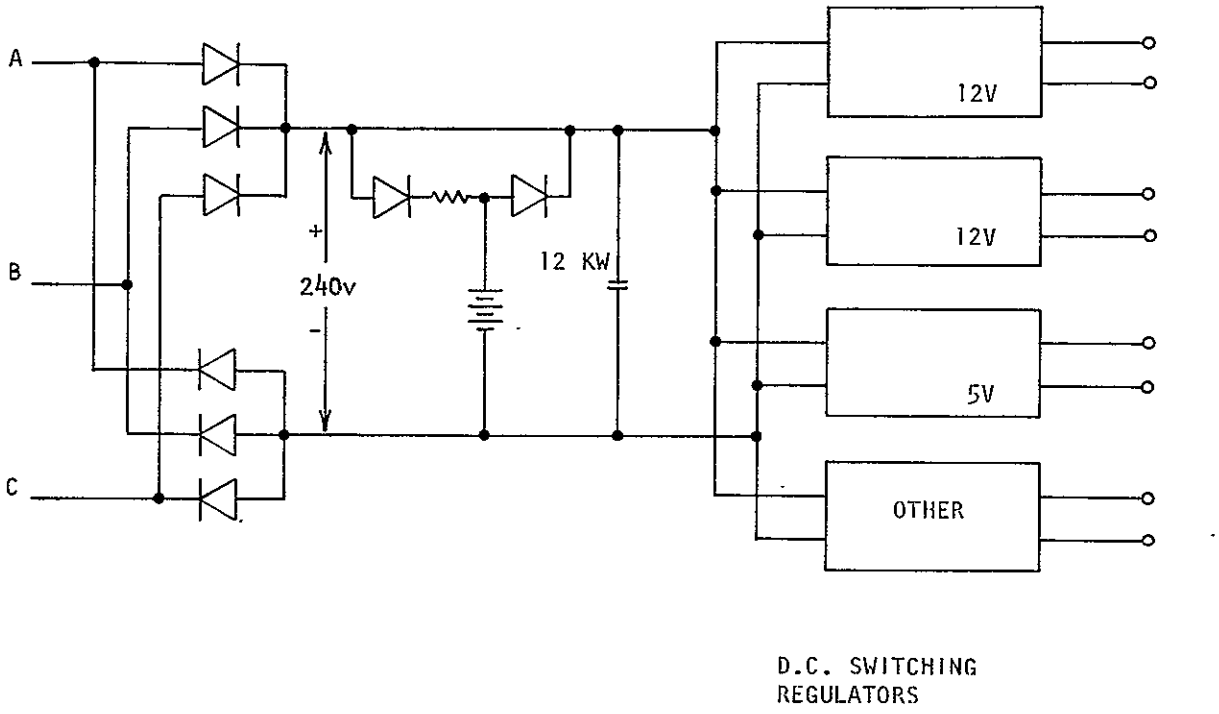


Figure 3-8. Uninterruptible Supply for the DBM (if CCD)

3.7 PHYSICAL DESIGN ISSUES

The physical design of high performance digital equipment is a subject that is often underrated in importance.

Subjects discussed include:

1. Signal distribution, among the bays, and along the backplane, is a special case because the wiring is particularly regular.

2. Power supply design is dependent on physical design for control of impedances, for reduction of noise on the power supply lines, and for reduction in electromagnetic interference.

3. EMI control is exercised at two levels. First, packaging and layout produces circuits that do not emit unwanted radiations. Second, shielding, primarily imposed by physical design, contains those radiations within the equipment.

4. Cooling is a major issue.

Concepts to be used in the physical design include:

1. Belts of wire for compact, controlled-impedance signal distribution

2. Batch termination of signals on those belts.

3. Elimination of discrete components

4. Use of low-impedance stripline for power distribution

5. Complete, solid ground planes in the circuit boards

3.7.1 Packaging and Layout

Packaging and layout options have not yet been finalized. The two high-usage boards, the processor board and the EM module, are each about the same package count as the boards used on the MSI version of PEPE (now at System Development Corporation, Huntsville, Ala.). Reevaluation of the wiring rules, and the segmentation of the board into partially independent areas, should bring some relief in terms of complexity. For these two boards, a 6-layer implementation is envisioned. The central two layers are a ground layer and a power distribution layer; the ground layer is a solid sheet, except for small holes, and is brought out to every third pad or so in the connector edge of the board. Each surface then carriers two signal layers.

The use of card-edge connectors is permissible and may simplify the module physical design. Reliable card edge connections require attention to design details that are usually sloughed over in low-cost electronic equipment giving card-edge connectors a bad name. Reliable high-density connections require pin-and-socket connectors. Here, however, the density is low, and reliable card-edge connections are possible, reducing costs, and possibly saving back-plane area.

3.7.2 Signal Distribution

Signal distribution will be by means of belts similar to those found in ILLIAC IV, with some differences.

Inter-unit signals can be either differential or single ended. In belts, differential signals require three wires per signal. There are two for each signal plus a ground between each signal for shielding and a ground at the edge of each belt. Single-ended signals take one wire per signal plus a ground between each signal and on the outside of the belt, or just two wires per signal. (Figure 3-9). Single ended signals are judged to be adequate. Hence, an 18-signal belt requires 37 wires. A 24-signal belt requires 49 wires. The signals listed in Section 3.2.5 will require the following belts:

1. 18-signal belts to and from the transposition network on both faces, to PE and to EM (8 data plus 1 timing, 2 directions).

2. 24-signal belt from processor to row fanout board. Two belts leave each fanout board and daisy-chain past eight processors.

3. 24-signal belt from cabinet fanout board to each row fanout board. There is no daisy chaining here.

4. 24-signal belt (four of them) from CU to cabinet fanout board.

There is a similar fanout tree carrying addresses and control to the EM modules. The widths of the belts in that fanout tree have not yet been identified.

Choices are limited for belt fabrication. Low dielectric constant is needed for minimum delay. This implies polyester insulation. Teflon is a second choice, electronically slightly superior, but subject to mechanical damage, specifically a cold flow phenomenon which can create delayed faults.

Characteristic impedances are fairly well limited to the range from 75 to 100 ohms, over the range of wire sizes from No. 28 to No. 32 on 0.025 or 0.050 inch centers.

All ground wires are carried through to the logic ground of the source or destination circuit at every connection to the belt.

Highly reliable termination to connectors is by batching soldering the belts. The paddleboards or other media to which the belts are soldered have etched copper conductors at cable-wire spacing, each conductor having a line etched down its center to help align the wires of the belt (Figure 3-10).

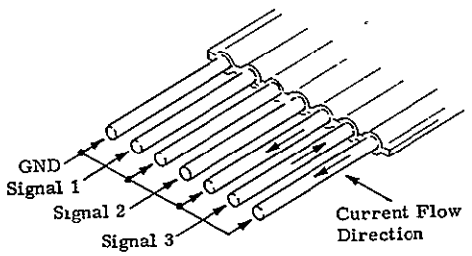


Figure 3-9. Unbalanced Signal

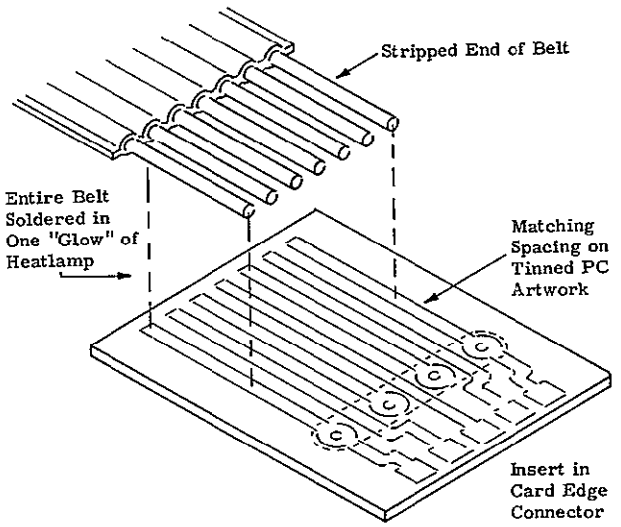


Figure 3-10. Paddleboards

3.7.3 Power Supply

The NSS may consume up to a quarter of megawatt of power from the ac mains, plus a comparable amount of power in the cooling equipment. It is important to design a power supply which is relatively efficient, as well as showing good regulation, tolerating lack of quality on the ac mains input and being efficient of space. This implies switching power supplies, and solutions to the resulting noise problems.

Many conventional "good practice" rules of thumb for grounding and power supply connections must be revised for high-current, low voltage systems. A correct rule is that no wire carrying large currents is allowed to be the connection whereby the ground "reference" voltage is tied together between various parts of the machine. Thus power supply returns are isolated from each other at the power supply end, eliminating ground loops. Backplanes are tied together through the ground wires accompanying signal cables. Backplanes are tied to reference ground directly, so the return wire at the power supply is not a reference point for ground. Reference ground is tied to chassis at one or more points. When both reference ground and chassis are strictly unpotential, as we expect in the NSS, multiple ties to chassis are allowable, and are a convenience in debugging.

Packaging individual power supplies with each processor has significant advantages. Instead of distributing hundreds of amperes at backplane distribution impedances of 0.0001 ohm or so, one can distribute unregulated high voltage at reasonable impedances, greatly simplifying the backplane construction. The independence and self-sufficiency of the processor is enhanced.

If individual supplies are not used, then hundreds of amperes must be distributed with voltage accuracies of tens of millivolts. Between supply and load, only a few nanohenries of inductance are allowed. To keep inductance down, the backplane is constructed as a stripline, and the output transistors of the shunt regulator and the card connectors are both mounted together on the same stripline (electrically, although a bolted connection allows mechanical separation).

3.7.4 Return Wires and Shields

Every signal (except for the very shortest signals), is carried on two conductors so that there is a return conductor, or single ground conductor, with every signal. Power supply connections are likewise made with both "going" and "return" conductors contained in the same stripline or pair of wires. Signal returns are tied to signal ground at both ends, to allow the return current to flow, to provide inductive shielding, and to eliminate mutual inductance. Such multiple ground connections are often erroneously called "ground loops". Differential signals are carried on 3-conductor cable.

3.7.5 EMI Control

The EMI (Electromagnetic Interference) requirements on the NSS have not yet been determined.

The design so far described will be free of many features that tend to promote electromagnetic radiation. For example, every signal is accompanied by a return wire so that large loops that tend to radiate are eliminated. The switching regulators are separated from the power mains by filters.

On the other hand, any EMI reducing feature that would cost significantly in terms of design, time, performance, or cost, has not been included. Totally shielded RFI-tight enclosures are not being proposed, although clearly they could be provided, at a price, if required. Isolation of signal ground (or "reference ground") inside the cabinets from the shielding enclosure is required for good EMI control. Here, it is proposed to connect signal ground to chassis at multiple points. Isolation will be a nuisance during debugging, may make the NSS more susceptible to its own interference, and will necessitate the including of transformer or optical isolation of signals.

3.7.6 Cooling

The NSS (excluding the DBM) may dissipate a quarter of a megawatt, as a crude estimate. There are nine or ten cabinets (four processor cabinets, four EM cabinets, and one or two cabinets for the transition network, control unit, and diagnostic controller). Thus, there is roughly 25 kilowatts per cabinet.

The quantity of heat involved is difficult but not impossible to handle by air cooling. Air cooling has maintenance advantages, since circuitry is not tied into the equipment by fluid-bearing umbilicals. Fluid cooling allows denser packaging with high-powered components. Heat pipes may be useful for transferring heat from specific hot spots. The final cooling design must be based on the packages in which the desired LSI components can be made available. Considering the rate at which new package types are beingintroduced in recent times, it would be premature to prejudge the cooling design at this point.

CHAPTER 4

SOFTWARE AND OPERATIONAL CHARACTERISTICS

4.1 MATCHING

An underlying concept in Burroughs computer development since the B 5000 has been the congruence between hardware and software design. In the present instance, the architecture of the machine has been matched to the class of programs that represent the problem, and the language is also matched to this same set of applications. The architectural features, such as the instruction set and the patterns of accessing the data, therefore match the language, or at least the normal usage of that language as it applies to the set of applications.

The following sections discuss the operating system, the language and compiler, the instruction set, and in a separate writeup, the data allocation.

4.2 OPERATING SYSTEM DESCRIPTION

4.2.1 Introduction

An operating system description, for the NSS, can at this time be reasonably complete only if it based on existing system software. The Burroughs B 7800 is a candidate for the host processor, and the attachment of the NSS to the host bears similarities to the attachment of the BSP to its B 7700/B 7800 host. The following

operating system description is based on using the B 7800 and the B 7800 Master Control Program (MCP), and upon leaning on some of the design decisions and experience gained in the BSP project.

The following description is therefore conditional on using a B 7800 for the host processor.

4.2.2 Objectives

4.2.2.1 Purpose of Operating System Software

The purpose of the operating system is to provide software support for the following:

1. Scheduling and controlling the flow of programs and files between the B 7800 and the NSS.
2. Allocating and maintaining the temporary files on the DBM.
3. Supporting the NSS FORTRAN programs for functions that cannot be performed in problem mode because of overall system implications.
4. Supporting other functions of the B 7800-NSS interface such as performance monitoring, error logging, and operator control.
5. Providing certain system utilities such as dump and log analyzer.

4.2.2.2 Workload Assumptions

The NSS is designed to operate efficiently on tasks with the following characteristics:

1. Data base sizes up to the size of the EM.
2. Long running programs: a minimum runtime of one second, a typical runtime of several minutes to several hours.
3. Batch job oriented.

4. 2. 2. 3 Salient Characteristics

A basic tenet of NSS design is to provide continuous execution of aerodynamic simulation programs at rated speed. To this end, Data Base Memory provides access time and transfer rates sufficient to sustain fully-overlapped sequential I/O, and provides capacity to contain all files required by a typical program on this high-performance device.

4. 2. 2. 3. 1 Computational Envelope

The computational envelope is defined as: "The NSS task, once started, runs to completion within the high-performance computational and I/O environment of the NSS without requiring intervention of or access to the much slower B 7800 processor or its I/O devices." In particular:

1. All NSS program and data files must be fully contained within DBM while the program is in operation; input and output files must be copied to or from DBM before the task is started or after it completes, respectively.

2. Each NSS program is self-contained as far as resources are concerned. No dependencies on B 7800 actions shall occur during the runtime of the program.

4. 2. 2. 3. 2 B 7800 Integration

NSS programs exist as tasks within the standard WFL (Work Flow Language) job structure of the B 7800. The job task itself runs entirely on the B 7800. A NSS task, once initiated, runs wholly within the computational envelope without further B 7800 dependence until it terminates.

4. 2. 2. 2. 3 Limitations

Some functions traditionally associated with operating systems are not provided on the NSS, although of course they are a normal part of theB 7800 itself. Specifically:

1. NSS FORTRAN is the only language provided.

2. Interactive programs are not supported.

3. Initially, no provision will be made for programs whose total file sizes exceed Extended Memory capacity.

4. Operator intervention on behalf of running aerodynamic simulations is disallowed, although the operator may terminate an NSS task.

5. A mix of tasks with very short runtimes can not be processed with maximum efficiency.

4.2.3 MCP Software

4.2.3.1 Overall Software

The MCP software consists of or interacts with the following components:

1. A compiler and binder for NSS FORTRAN programs.

2. A modified version of the B 7800 MCP (Master Control Program) and WFL compiler.

3. A supervisory control program for the NSS.

4. A network definition that includes Burroughs-supplied NDL (Network Definition Language) source to support the B 7800 Data Communications Processor (DCP) to Diagnostic Controller (DCP) interface.

5. A maintenance and test routine which runs on the B 7800 using DCP-DC interface.

6. All other standard B 7800 software selected by Ames.

7. Various utilities and subroutine libraries.

Items 2., 3., 4., and some of 7. constitute the NSS operating system.

4.2.3.2 Organization

The MCP software consists of two major parts: (1) modifications and extensions to existing software that runs on the B 7800, and (2) new software to run on the NSS.

4.2.3.3 B 7800 Software for the NSS MCP

Modifications and extensions to the B 7800 MCP and utilities will provide the following functions:

1. Initialize the NSS system.
2. Process the extended work flow language for aerodynamic simulation jobs.
3. Allocate space in DBM and maintain the DBM directory.
4. Transfer files between B 7800 devices, including archive, and DBM.
5. "Forward" tasks to the NSS (DBM) for processing and retrieve them when completed.
6. Maintain a log of performance, error, and accounting information.
7. Provide various "analyzer" utilities to edit and format dumps, logs, etc.
8. Schedule and initiate tasks on the NSS.

4.2.3.4 NSS Software

MCP software on the NSS will provide the following functions.

1. Interface to corresponding B 7800 software for NSS initialization, task forwarding, dump, etc.
2. Provide wrap-up for normal and abnormal termination.
3. Provide processor program overlay loading for NSS FORTRAN programs.
4. Service NSS interrupts, such as those errors that cannot be handled by the individual processors.

4.2.4 Other System Software

4.2.4.1 Job Structure

4.2.4.1.1 Introduction, WFL and the Job Concept

A job is a program (and task) which invokes, and determines the relative sequence of, a set of programs. These programs constitute a set of logically related tasks which perform sone data transformation on files. A job is written in NSS Work Flow Language (WFL) and it runs on the B 7800. The WFL contains the standard B 7800 WFL as a proper subset, so any B 7800 job can run unmodified on the host of the NASF. The NASF extensions in WFL provide the following functions.

1. Invoke NSS FORTRAN compiler and linker.
2. Specify resource requirements for scheduling and allocation purposes.
3. Copy files between B 7800 devices and DBM, and specify relevant file attributes.
4. Specify a job restart point following B 7800 or NSS failures.

4.2.4.1.2 Organization of a Job

The basic outline of a typical job is constrained by the computational envelope and LINKER concepts. The typical job will contain, in this sequence:

1. None, one, or more NSS FORTRAN compilations.
2. A LINKER task.
3. One or more COPY tasks to transfer input data files from B 7800 to Data Base Memory.
4. A COPY task, is needed, to transfer program to CUM (Control Unit Memory)
5. One or more executions of NSS programs.
6. One or more COPY tasks to transfer the output files back to B 7800 devices from Data Base Memory.

In addition, any number of B 7800 tasks may be interspersed with the above, such as to generate input files or process output files. These are asynchronous with, and do not impede the execution of other NSS tasks.

4. 2. 4. 2 FORTRAN Support

Implementation of I/O support is divided between the B 7800 MCP, the NSS MCP, and the compiler-generated program code. B 7800 MCP code will copy the files between DBM and B 7800 devices. The SCHEDULER on the B 7800 will provide DBM file allocation and directory maintenance, and connect files to programs. NSS code will process error interrupts, and NSS problem code will perform actual transfers between DBM and extended memory, including DBM address, EM addresses, and sending control words to the DBM controller.

The following FORTRAN functions will be supported by NSS software:

1. STOP will transmit the operand to B 7800 supervisory console, and terminate with normal End of Task (EOT).

2. Error End of Task.

Checkpoint will not be supported. Restarts, in aerodynamic simulations, are more suitably an applications program responsibility.

4. 2. 4. 3 Program Load and Overlay Support

The NSS is designed to run only one program at a time. No additional program or data area may be preloaded into CUM, PEM, or PEPM in order to minimize setup delays when starting the next task.

The LINKER accepts object code files from one or more separate FORTRAN compilations and produces a single load code file, called the loadfile. In the process, the LINKER assigns memory locations to all program instructions and data areas, and resolves or relocates address references accordingly.

For the case that the PEPM part of the user program is too large to be completely PEPM contained, the LINKER supports an overlay facility. With this mechanism, the user may divide a program into multiple phases, and specify which phases may share the same memory locations. The LINKER must also generate certain information for the MCP.

The loadfile for an NSS task contains information to be used by the B 7800 MCP for forwarding, the NSS code (if different from the previous task's), data, and a directory containing descriptive information about the code and data.

Data is either initialized, uninitialized, or initialized to "invalid". Initialized segments have their initial contents present in DBM as generated by the Compiler/Linker. Uninitialized segments occupy no space on DBM, linkwise segments to be initialized to "invalid" are not present on DBM, but are initialized to "minus infinity" by the execution of appropriate NSS code.

4.2.4.4 System Operations

Information will be accumulated if the "monitor" option has been selected at run time. Some of this information is accumulated by the B 7800 as part of normal monitoring; some by the NSS-resident operating system. The following is a list of information that may be included in such monitoring.

1. Interval timer reading at time of report (for NSS information).
2. Real time Clock at time of report (in B 7800).
3. Count of TN-using instructions.
4. Some measure (to be determined) of PE idle time.
5. A measure of CU idle time.
6. Count of error corrections performed.
7. List of addresses associated with error corrections.
8. Time spent in specific subroutines (CU subroutines, at least; PE subroutines present a monitoring problem, since time can be different in each processor.)

The CU's interval timer is coordinated with the B 7800 clock at the beginning of a run.

The MCP will log, for the NSS, the same types of actions presently logged by the B 7800 MCP, including Beginning-of-Task and End-of-Task of NSS tasks, OPEN and CLOSE of DBM files. The B 7800 log will be extended to include records of forwarding action and records of COPY action to/from DBM. Status, as displayed on the B 7800 operator's console, will be extended to include NSS tasks.

NSS Initialization is that process whereby the NSS is transformed from any state into the state in which it normally processes user programs. This process re-initializes all parts of the system. No process corresponding to B 7800 "coolstart", where the disk directory is saved, or to a B 7800 "halt-load", where jobs are restarted from a last inactive point, is to be implemented on the NSS. Although the NSS MCP does not preserve task state across failures, the job can be restarted via B 7800 intervention, including restarts from the restart points provided in the application program itself.

Initialization consists of the following steps:

1. The driver program, running on the B 7800, determines that the B 7800-NSS DC connection is operational.

2. The driver transfers NSS MCP to the CU via the DC. Once this piece of code is loaded into CUM, the NSS is started running on its own.

3. An initialization phase of the NSS MCP will perform various initialization functions, including confidence tests.

4. The MCP will then complete its initialization, and inform the B 7800 via the CU-B 7800 interface

The NSS will then be ready to process programs.

4.3 LANGUAGE AND COMPILER

The source language that takes full advantage of the parallelism of the SAM will look to a large extent like normal FORTRAN, with extensions and with some restrictions.

The compiler and its language perform a number of functions:

1. Allows the user to control the machine.

2. Converts user inputs into machine language.

3. Transforms the serial input language into parallel operations on the array.

The interface with the user (the language) has a number of features:

1. It is FORTRAN-based (FORTRAN with extensions),

2. Convenient constructs are provided to allow the user to obtain full use of the power of the machine.

3. The user is not asked to specify information that the compiler can just as well determine for itself.

4. Error messages are provided to alert the user to erroneous input. The compiler will not abort on error, but continues to compile for syntax.

5. Warning messages alert the user to possible inefficiencies in his compiler code.

The compiler performs part of the analysis, leaving the user free to concentrate on the problem at hand. For example:

1. The parallelism specified by the user is problem-sized, independent of the number of PE's (512). The compiler takes the parallelism inherent in the problem and cuts it up into 512-sized pieces for execution.

2. The compiler distinguishes between CU DO loops and PE DO loops on the basis of the content of the loop.

3. The compiler distinguishes between PE subroutines and CU subroutines on the basis of the content of the subroutine.

4. The compiler takes care of address calculation in EM on the basis of the data declarations. Explicit computations related to EM addresses are never required in the program source deck.

5. All details of the transposition network functions are hidden from the user.

For the machine, the compiler performs certain efficiency-preserving operations:

1. PE index register usage is designed to minimize the need for storing indices or other integers in PEM. When two or more variables in the same statement are indexed by the same set of indices, the offset is calculated only once.

2. PE floating point register usage is based on an algorithm that minimizes PEM fetches and stores. Variables created in one statement are left in registers if needed for the next statement, not refetched.

3. Opportunities for using the multiply-add instruction are recognized and exploited.

A key point in the design of language and compiler is to recognize that the users of the NSS are primarily aerodynamicists, mathematicians, and numerical analysts more than they are programmers. A great deal has been learned in the last twenty years about how to present the necessarily complex machinery of a computer to the user in such a way that the managing of the complexity is automated and hidden, and the user is presented with a tool that bears directly upon his problem. In the present instance, for example, the number of PE's, 512, shows nowhere in the input language, and programs can be written calling for any degree of parallelism at all.

The extensions to the language (Table 4-1) reflect the architecture of the NSS. Somewhere inside a program from the NSS, one may find statements such as:

```
DO PARALLEL 10 J = 1, 100
DO PARALLEL 10 K = 1, 27
      (statements)
10 CONTINUE
```

The statements inside this loop will be done within the PE's. The compiler will assign a specific pair of J, K values to a specific PE. Within the statements, then, statements are executed over 512 sets of J, K values simultaneously, and the J, K indexing operation is implicit in the PE number of the PE that is executing on any particular piece of data. One advantage of the NSS is therefore the elimination of some of the indexing necessary in a conventional serial machine and replacing it by place-of-execution.

The compiler will take the programmer's word that his DO PARALLEL declara represents a valid parallelism. If some operation within the loop(s) cannot be executed in parallel across the parallel indices, this is an error.

Analysis of the codes submitted by Ames shows that almost all of the computation falls into repetitions of the following typical pattern:

1. From a data-base array, which would be declared to be in EM, a set of variables are fetched, which are the inputs to a parallel computation. Typical operation is in parallel on two dimensions of a three or more dimensional array.

2. The resulting variables are the input to a fairly extensive amount of computation which does not involve any other variables.

3. The results are stored back into EM.

Table 4-1. Preliminary Definition of Extentions to Normal FORTRAN for NSS FORTRAN

	Definition
DOPARALLEL	Equivalent to DO, but tells the compiler that each iteration of the loop is independent of the previous iteration, and that therefore all iterations can proceed simultaneously in parallel, no limit is placed on the number of iterations. Temporary variables within the iterations are undefined upon exit from the DOPARALLEL, except for those that receive the result of global operations.
EM ARRAY	A declaration similar to the DIMENSION statement. Variables in EM ARRAY's are saved upon exit from DOPARALLEL loops. EM ARRAY declarations must precede any DOPARALLEL loop in which they are used.
EM ARRAY A(...)=B(...)	The indices in A are a reordered subset of the indices in B, and the ranges are also a subset. An equivalence declaration which results in the reordering of indices in order to avoid writing multiple copies of almost-identical code.

Table 4-1. Preliminary Definition of Extentions to Normal FORTRAN for NSS FORTRAN (Cont'd)

	Definition
CU Variables	Variables stored in the CUM, where a single variable is used across the entire extent of the parallelism, such as for controlling synchronous DO loops, will probably need to be explicitly declared. Such declarations will generate "WARNINGs" about efficiency.
PACK and UNPACK	Intrinsics to pack two 24-bit words per 48-bit word, or to unpack them.
ENDDO	Optional substitute for labelled "CONTINUE" statements.
GLOBAL	GLOBAL operations are called from within DOPARALLEL loops. These include maximum, minimum, sum, product. Although the input language makes these operations look like functions, they are macros. Results can be left in a CU variable, or in a PE variable.
DO	DO will be a CU (global) or independent PE loop depending on the variable used as loop control.
SUBROUTINE	A distinction is made between a synchronized or "CU" subroutine and an unsynchronized or "processor" subroutine. The compiler, when compiling the calling program, must have some way of identifying the type of subroutine being called. Since separate compilation of subroutine is allowed, apparently two types of calls will be needed. Further discussion of this point is needed.

Indexing patterns are generally simple. Three nested DO loops on the geometric variables, with a small fixed-length DO loop inside the third index, cover most cases. Operations inside the outer two loops are independent of previous or following iterations, so that, in a 100 by 100 by 100 grid, for example, one can call for the 10,000 iterations of the inner loop to proceed in parallel.

```
      EM ARRAY A(18, 100, 100, 100)
      DIMENSION B(18, 100)
          (other declarations or statements intervene here)
      DO PARALLEL 15 K=1, 100
      DO PARALLEL 15 L=1, 100
      DO J=1, 100
      DO N=1, 18
      B(N, J) = A(N, J, K, L)
      ENDDO
      ENDDO
      DO J=1, 100
          (here a sequence of arithmetic statements)
      ENDDO
      DO J=1, 100
      DO N=1, 5
      A(N, J, K, L) = B(N+2, J)
      ENDDO
      ENDDO
15    CONTINUE
```

Figure 4-1. Example of Typical Pattern in Simplest Form

The typical pattern can be programmed as shown in Figure 4-1. The entries have the following significance.

1. EM ARRAY declares that the following array is resident in EM; that is, it is a database array. Fetches from and stores to an EM ARRAY will be via the transposition network, so that, in the case of three geometric variables J, K, L parallelism may be present on any two of the three indices.

2. Dimension statements declare arrays within PEM.

3. DO N=1, 18 followed by B(N, xx) = A(N, xx) will result in a single EM fetch of 18 words, since the inner loop N is the first index, and addresses are sequential in PEM. If the order of looping were reversed a "WARNING, INEFFICIENT CODE" message would be produced.

4. ENDDO is an alternative to CONTINUE statements for ending DO loops.

5. A(N, J, K, L) = B(N+2, J) stores results back into the first through fifth elements of A from the third through seventh elements of B.

To share code while indexing along different grid dimensions, one needs to not only call subroutines using different indices, one also needs to fetch data from A to B (and store it back) where the index in B is first one, then another, index in A. The simplest method, from the point of view of least compiler complexity used is shown in Figure 4-2. Separate EM read statements, each looped on a different index J, K, and L, precede calls on PE subroutine DOIT. Inside DOIT is a loop on the index that is passed as a parameter to DOIT.

A second mechanism for sharing code (used in the BSP extensions of FORTRAN) is to declare arrays with the indices permuted. A description of the index permutations is then passed to the subroutine when it is called, and used to control the different modes of fetching. Figure 4-3 is an example. Explanations of various lines are:

D(18, J==:100, L=1:100, K=1:100) = A(18, J, K, L) declares that D, when referred to is the 100 by 100 by 100 subset of array A with indices permuted first, third, second.

```
      EM ARRAY A(18, 100, 100, 100)
          (declarations)
      DO PARALLEL 15 J=1, 100
      DO PARALLEL 15 K=1, 100
      DO 25 L=1, 100
      DO 25 N=1, 18
      B(N, L) = A(N, J, K, L)
25    CONTINUE
      CALL DOIT(J)
          (more statements, then:)
15    CONTINUE
      DO PARALLEL 16 J=1, 100
      DO PARALLEL 16 L=1, 100
      DO 26 K=1, 100
      DO 26 N=1, 18
      B(N, K) = A(N, J, K, L)
26    CONTINUE
      CALL DOIT(K)
          (more statements, then:)
16    CONTINUE
      DO PARALLEL 17 K=1, 100
      DO PARALLEL 17 J=1, 100
      DO 27 L=1, 100
      DO 27 N=1, 18
      B(N, L) = A(N, J, K, L)
27    CONTINUE
      CALL DOIT(L)
          (more statements, then:)
17    CONTINUE
```

Figure 4-2. Use of Source Code with Parallelism on Different Indices

```
      SUBROUTINE FACIT (IA, IB, IC, G)
      EM ARRAY G(18, 100, 100, 100)
          (declarations)
      DO PARALLEL 10 IA = 1, 000
      DO PARALLEL 10 IB = 1, 000
      DO 15 IC = 1, 100
      DO 15 N = 1, 18
      B(N, IC) = G(N, IA, IB, IC)
15    CONTINUE
      DO 20 IC = 1, 100
          (here, arithmetic operations within the PE)
20    CONTINUE
      DO 25 IC = 1, 100
      DO 25 N = 1, 5
      G(N, IA, IB, IC) = B(N, IC)
25    CONTINUE
10    CONTINUE
```

(a) Declaration Subroutine FACIT

```
      EM ARRAY A(18, 100, 100, 100)
      ARRAY B(18, K=1:100, L=1:100, J=1:100) = A(18, J, K, L)
      ARRAY D(18, J=1:100, L=1:100, K=1:100) = A(18, J, K, L)
          (other statements, then:)
      CALL FACIT(J, K, L, A)
          (later:)
      CALL FACIT(J, L, K, D)
          (later:)
      CALL FACIT(K, L, J, B)
```

(b) Use of Subroutine FACIT

Figure 4-3. Alternate Method of Using Identical Code on Different Indexings

- FACIT(J, L, K, D) then declares that the code FACIT is to be executed using K for internal indexing, and the array description D for controlling operations on the EM array declared within that subroutine.

Restrictions to normal FORTRAN are seen, but are not ominous. Inside a DO PARALLEL loop, the compiler takes the expressed parallelism and produces a hidden loop that uses the 512 PE's in several iterations to cover the entire parallelism. Thus, PE variables will be overwritten in a later iteration, and will be undetermined upon exit from the loop. Exiting from a DO PARALLEL loop thus wipes out all PE variables used within that loop except those that had the same value in all PE's (for example, a global maximum could be saved, or the last value of some index, but not an entire array full of values).

Equivalencing inside EM ARRAYS is restricted because of the need to perform EM fetches and stores efficiently. See the section on Data Allocation below.

4.3.1 Input/Output Operations

Several operations within the NSS bear some similarity to the I/O operations of a conventional computer.

Loading and unloading PEM from or to EM occurs in response to a CU-issued command after bringing the array into synchronism.

Dumping and retrieving snapshots and restarts from the DBM is done by means of commands issued from the CU. The CU needs the same addressing machinery to access extended memory for these as for loading and unloading.

4.4 INSTRUCTION SET

4.4.1 Code Emission from the Compiler

The compiler parses the input code, recognizes programmer-specified parallelism, finds additional parallelism by analysis, and translates the input source into a parallelized form. The analysis contained in the compiler will find parallelism for some serial-looking code that the programmer knew could run in parallel and parallelizes it; the purpose being that the programmer will be allowed to write

in a more familiar-looking language. It will not be guaranteed, on its own, to turn any kind of serial FORTRAN into efficient code for the NSS.

The compiler turns the sequence of statements in the input into a sequence of instructions for the NSS. There are three categories of instructions. First, there are those, like floating point arithmetic, that are executed within each processor independently of any other processor or the CU. Second, those executed in the CU without regard for what the processors may be doing. Third, those instructions which require cooperation between the CU and the processors for their execution.

Out of the compiler come two instruction streams. The first is the CU instruction stream; the second is the instruction stream for all the processors. Instructions in the third category occur in both streams, and include the requirement that all elements, the CU and all the PEs currently executing, are finished with previous instructions so that they can cooperate in this one.

It may be instructive to trace the source code through the compiler and see the resulting emitted code. Figure 4-4 shows part of the source code of a hypothetical program written in a hypothetical source language. Line 1 declares a 16 element array that is allocated space in every PE memory. Line 2 declares an array that will have 60 elements in every PE memory, and whose other dimensions will correspond to the dimensions over which parallelism is to be found. Line 3 declares a data base array in extended memory. In the declaration, it is necessary to identify the special indices over which parallelism is to be found, because of the compiler's need to pad such arrays to proper size. There may be some other way to state this language extension. Other declarations are skipped over.

Lines 4 and 5 says that 10,000 iterations over the J and K indices can be done in parallel. Since there are only 512 PEs, and since the memory allocation algorithm divides the array up into subarrays, to be specific in the example, say measuring 5 x 100, the compiler must implement a loop that iterates 20 times, since it takes a set of 20 of these subarrays to cover the declared A array. DO loops, cut the declared 10,000-wide parallelism up into PE array-sized pieces.

```
                                                          Line No.
   DIMENSION G(4, 4), F(100, 4), AA(100, 4, 4)              1
   DIMENSION B(3, 20)                                       2
   EM ARRAY A(20, J=1:100, K=1:100, L=1:100)                3
         .
         .
         .
   DOPARALLEL J=1, 100                                      4
   DOPARALLEL K=1, 100                                      5
   B(1, 20) = A(20, J, K, 1)                                6
   B(2, 20) = A(20, J, K, 2)                                7
   DO 6 L=1, 100                                            8
   IF(L. LT. 99) B(3, 20) = A(20, J, K, (L+2) MOD 3)        9
   IF(L. GT. 1) DO                                          10
   DO 14 N=1, 4                                             11
   F(L, N) = F(L, N) -AA(L, N, 1)*F(L, 1) -AA(L, N, 2)*F(L-1, 2)   12
       - AA(L, N, 3)*F(L+1, 3) -AA(L, N, 4)*F(L-1, 4)       13
         .
         .
         .
14 CONTINUE                                                 14
         .
         .
         .
   ENDIF                                                    15
   CALL FILTRZ                                              16
   CALL LUDEC(G)                                            17
   CALL FORBAK(G)                                           18
 6 CONTINUE                                                 19
         .
         .
         .
   ENDDO                                                    20
   ENDDO                                                    21
```

Figure 4-4. An Example of Source Code

Line 6 is a fetch through the transposition network from the A array to the B array. So is line 7. These two lines provide the loading of the first two elements of a three element circular buffer to hold the data base while L is iterated. Line 8 iterates on the third dimension. Depending on whether L has been declared a CU variable or a PE variable, this will be a CU test, synchronized across the entire array, or a PE test, independently done in each PE. In the present case, it does not matter, since the next line will synchronize the DO loop whether the loop statement itself does or not.

In line 9, L is not within 2 of the end of the L loop. Another section of data base must be fetched to the circular buffer. The fetch from extended memory will result in synchronization of the array. Line 10 is either a CU test or a PE test depending on the declared place of residency of L. As with line 8 and 9, it will not matter this time because the array is freshly synchronized as a result of the fetch from extended memory in the previous line. Resynchronizing wastes virtually no time compared to the execution time of the rest of the DO L loop. Line 11 ought to be, for efficiency, a PE test. The programmer should remember to declare N as a PE integer.

Lines 12 and 13 are a typical PE arithmetic statement. F has been declared as a PE array, as has AA. Line 14 marks the end of the DO 14 loop on N. Line 15 marks the end of the IF condition of line 10. The branch instructions associated with this point are either PE or CU depending on L again. Lines 16, 17, and 18 are Subroutine calls. For best efficiency, these should have been declared as PE subroutines. Perhaps the compiler can recognize PE subroutines from the array and variable declarations. Line 19, CONTINUE, is the end of the L loop, a serial loop. Lines 20 and 21 are the end of the parallel J, K loops, and therefore are also the end of the hidden loops that processes the 20 iterations.

Figure 4-5 shows the resulting code streams, in very generalized form, of PE and CU instructions. Double arrow-head lines connect the points of synchronism in the two data streams. The L index is assumed to be a PE index. Statement numbers are used to key the codes of Figure 4-5 to the statements of Figure 4-4.

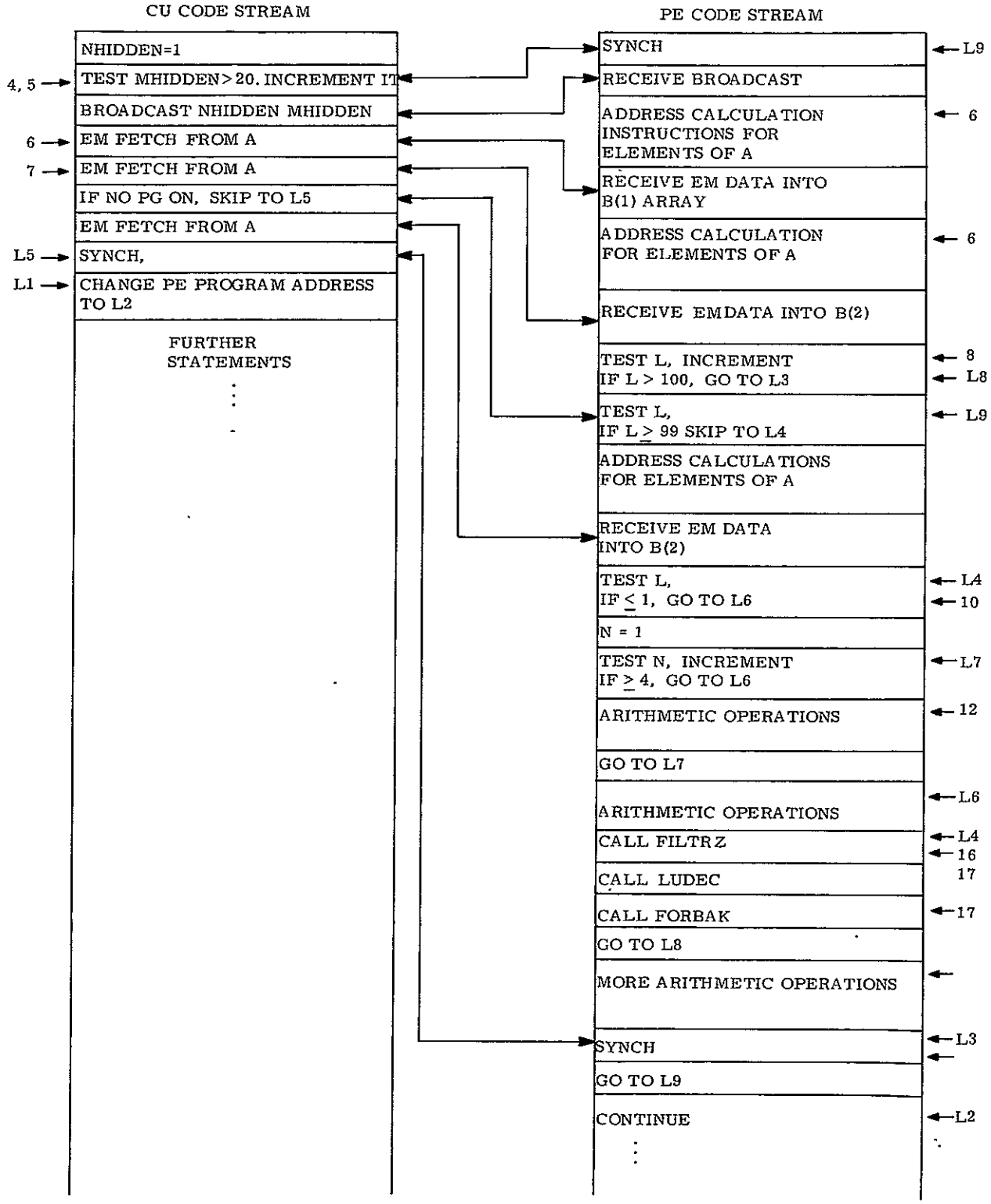


Figure 4-5. Matching Code Streams

4. 4. 2 Instruction Set Tables

Table 4-2 lists the instructions in the Navier-Stokes Solver. The instructions occur in two separate instruction streams, which are compiled together. One instruction stream contains instructions for the processing element; one contains instructions for the control unit. Table 4-2 therefore contains three sections. The first is those instructions which are self-contained within the individual PE, such as floating point multiply. The second section section lists those instructions in which are executed entirely from the CU. This includes those instructions in which the CU forces all the PE's to take a particular action that overrides the current PE instruction. The third section of the table includes those instructions which occur as matching elements in both instruction streams. "Synchronize", for example, is an instruction found in all instruction streams. Only after all PE's have reached this instruction will the CU cause the next operation to be executed simultaneously in all PE's. All of the instructions in the third section require some sort of synchronization, since all elements of the NSS execute it jointly.

The mechanism of synchronization is described in the section on the control unit. In a type I synch the PE's wait for the control unit to say "go" before doing the instruction. In a type II synch, the control unit performs an action (say setting the TN), and raises an "all is ready" line. PE's see this line up and go on. The CU will wait to go on until all "I got here" lines are up.

4. 4. 3 Discussion

4. 4. 3. 1 Control Bitvectors

Each processor has at least two control bits. The first, an "enable" bit, says whether that processor is engaged in the current computation. The second bit is the "run" bit. When set, it says the processor is "running"; when reset, it says the processor is "waiting". These bits are used to help implement the synchronization process. For example, the CU will not issue a "go" signal for a synchronous action until all enabled processors are waiting.

Table 4-2. Processing Element Instructions

Arithmetic Instructions (Floating Point Operands)	
ADD, SUB MUL, DIV	One operand is in memory, register, or is literal. Second operand is in register. Result to register. Literal option allows MUL by 1/2, ADD ±1, reciprocal, etc., in one instruction.
MAD	Add product of two operands to third. Leave result in register.
SSQ	Form sum of squares of two operands.
ABS	Make sign positive.
ADDD, MULD	Double-length sum, Double length product.
Index Arithmetic (Integer Arithmetic) See note 1. Both single (16-bit) and double length (32-bit) 2's complement	
IADD, ISUB	Normal integer arithmetic. Literal option saves instructions.
IMUL	Integer multiply
IDIV	Integer Divide.
ID521	Fast DIV by 521. Used in calculating EM addresses.
IMOD	Saves remainder instead of quotient from integer division.
ILIT	16-bit literal to integer register.
Test and Branch	
LT, LE, GT GE, EQ, NE	Normal two-way tests. Proper response if one or both numbers are "infinity" to be determined. "Infinitesimal" is smaller in magnitude than any number except zero.
TIX	Test and increment indices. Non-unity increments permitted.
AND, OR, NOT	Logic combinations of tests results, and of the one-bit registers, one of which is the enable bit.
BIT	Test LSB of 16-bit word (used for this processor's bit of the 521-long bit vector).

Table 4-2. Cont.

JUMP	Branch Unconditionally.
CALL	Subroutine entry. Includes automatic handling of the parameters to be passed to the subroutine, including the array descriptors. There is a LIFO stack of subroutine entry and return information. This stack does not allow recursive calls, since the addresses within the code file are in absolute.
RETURN	Return information, at the end of the subroutine, will be found on the top of the subroutine return information stack. In the case of a function subroutine, the top of the stack is also used as the location for the value to be returned. The stack is in memory (PEM). On return, the information for this subroutine is deleted, and the stack is cut back.
INFY	Test for equal to "infinity".
INFL	Test for equal to "infinitesimal".
POP	Execute stack action of "RETURN", but do not change program counter setting. Expected to be useful in disgnostics.
TOS	Set stack pointer to new value.

Conversion and Formatting

FIX	Integerize a floating point number, destination will be integer register.
FLOAT	Take contents of integer register, convert to floating, and insert in floating point register.
INFZ	Convert operand to zero if infinitesimal.
SETFL	Set infinitesimal control bit. Underflow creates infinitesimals.
SETZ	Reset infinitesimal control bit. Underflow creates zeros.
PAK2	Round two 48-bit floating operands to 24 bits each, and pack into one 48-bit register.

Table 4-2. Cont.

PAKI	Pack two integers into two 24-bit fields on one 48-bit register.
MIM, MIL	Move the most significant (or least significant) half of a 48-bit word to a 32-bit integer location, in the least significant 24 bits.
MFL	Move the least significant half on one 48-bit register to the leading half of another, with zero fill.
ZFM	Zero the least significant half of a 48-bit (floating point) register.
Memory and Data Movement	
FETCH	From source desginated in address field to register. Source may be register, PEM, or literal.
STORE	From register to destination designated in address field. Integers are stored right justified with zero fill.
System Control	
WAIT	Instruction of indefinite duration. Resets "run" bit while leaving "enable" set. Will be exited when CU sets program counter to new value, or on time-out interrupt.
STOP	Reset "run" and "enable" bits both. Do not fetch next instruction.
HELP	Reset "run", and "enable" bits. Send ONE on interrupt line to CU.
PNO	Read processor number into specified integer register
SB	Set LSB of word to result of test in previous instruction
SHF	Shift right end around N places (48 bits)

Note 1. Almost all integers need be only 16 bits long. Only a few of the integer operations are used for the 32-bit EM addresses. Normal integers are 16 bits, but double-length integers will be used for EM addresses. Thus, each instruction comes in two forms, single length and double length. ID521 need be implemented only in the extra-precision mode.

Table 4-2. Cont.

CU INSTRUCTIONS

(The CU instruction list is still incomplete)

System Instructions

Test interrupt register (automatically resets bit)

Set/Reset bit in mask register

CU Register Instructions (used for address calculations, TN settings, etc.)

Integer arithmetics, IADD, ISUB, IMUL, IDIV, IMOD,

Numeric shifts

Jump Instructions

In user programs, there may be no need for CU branch instructions other than the capabilities found in the list of Joint-CU-PE instructions. However, for diagnostics and initialization, there is need for CU branching independently of any cooperation of the PEs. Presumably the same logic tests are available here as in the Joint-CU-PE list, plus unconditional Jumps.

I/O

DBM-EM transfers. Load and Unload. Operation requires the specification of DBM starting address, EM starting address, and number of words. For details of this operation, see the section below on "Data Allocation".

Transmit word to host, receive word from host.

Unconditional Control of PEs (requires no necessary PE state)

Broadcast to PEM or PEPM. Increment address register in PE for each word received.

Set PE register, including PCR (Program Counter Register), PEMAR (Processing Element Memory Address Register)

Set "on/off", "run", etc., bits in PEs

Halt at end of next PE instruction; Restart PEs from halted state

Halt unconditionally, in mid-instruction.

Table 4-2. Cont.

EM Access

LOADCU. One of the otherwise unused ports of the TN (the 513th, say) is used to transfer data to the CU from the module selected by the offset.

STORECU. Similar to LOADCU except for the direction of transmission of data.

JOINT PE-CU INSTRUCTIONS

(An instruction is found in both the CU and PE streams)

Transposition Network Instructions

LOADEM. Fetch from EM to PE Register or PEM. In the PE instruction is the address of the register containing the EM address, and an address field (which may contain PEM address or register address), and the address of the register containing the increment. In the CU instruction are three register addresses, pointing to the offset and skip distance settings for the transposition network, and the number N of words to be transferred. N will be broadcast to the processors during the execution. Synch is type II, for timing purposes.

STOREM. Similar to LOADEM except for the direction of transmission of data.

SHIFTTN. In this instruction, the processor-to-EM and EM-to processor paths are set to different settings and at the EM side the output of the TN (going toward the EM) is wrapped around to feed the input to TN that is normally from EM. The result is a processor-to-processor data transfer corresponding to the two settings. For example, with both skip distances set equal to 1, the result is a routing equal to the difference in the two offsets.

EMNO. Read EM module number to the processor.

SET TN CONTROLS (may be needed for diagnostics).

Branch Instructions executed jointly

Tests on CU register contents, including index test and increment. Synchronous version of PE index test and increment. If any PE iterates again through the loop, the loop is taken, non-iterating PEs wait.

Subroutine CALL and RETURN, where array is synchronized.

Table 4-2. Cont.

Other

Synchronize (type I), variants of this instruction may modify the settings of the PE "on/off", "run", etc., bits.

Broadcast. Transmit one 48-bit word from the CU to all PEs that are on.

Unbroadcast. Read to the CU the OR of one 48-bit word from each PE that is on.

Note: Control bitvectors are stored in EM, and do not require any special CU instructions.)

DIAGNOSTIC CONTROLLER INSTRUCTIONS

(These instructions are passed one at a time to the DC via its own connection, separate from the CU's host connection. A maintenance panel issues these same instructions either via push button or perhaps via an incremental magnetic tape cartridge.)

Load CU Register (an appropriate subset of all CU registers)

Load words in CU memory at address A. (Address register is restored subsequently)

Execute "I", where I is a CU instruction. PE testing may involve the PE-controlling instructions loaded thusly into the CU.

Read CU register (approximately the same subset as for Load)

Step entire system 1 clock pulse. Note that the PE and EM clocks are delayed from the CU clocks by the wire length, so that if one clock sets a signal in the CU for the PE to read, the next clock will load that same data into the appropriate PE register. That is, the wire from CU to PE is of zero length with respect to clock phase.

Step 1 instruction in the CU instruction stream and halt.

Run N clocks.

Halt CU after next instruction.

Halt CU after next clock.

Run.

The CU can turn disabled processors back on, when the degree of parallelism expands again. This mechanism could be a bit-vector distributed from CU. Alternatively, the CU could turn on all processors, which then test bit vectors fetched from EM.

A second area where the design remains to be finalized is the case that some processors are to be loaded from one area stored in EM, and other processors are to be loaded to another. There will be two LOADEM instructions in the CU instruction stream. Some processors must cooperate with the first, others must cooperate with the second. The case that must be designed against is that processors waiting for the second somehow cooperate with the first LOADEM instruction. The simplest solution is a programming restriction prohibiting conditional execution of statements that compile to LOADEM or STOREM instructions. All enabled processors cooperate in every LOADEM. This costs no efficiency, since the two LOADEM instructions must be separately executed anyhow.

4.4.3.2 PE Registers

Registers in the PE are clearly called for. It is not desirable to store every temporary result into PE memory only to be refetched to the PE. Two recipes for supplying those registers have become popular. Recipe number 1 is the Polish stack, extolled at length in the May 1977 issue of "Computer" magazine; recipe number 2 has registers with addresses assigned by the compiler for every temporary variable.

Recipe number 2, addressable registers, gives the compiler more options about holding operands fetched in one statement so they can be held locally instead of being refetched in the next statement. Addressable registers are used at the expense of code file size, compiler running time and complexity, but allow the compiler to optimize throughput. Therefore the PE instructions are based on the use of registers addressible by the compiler.

4.4.3.3 Arithmetic Tests

In the codes that were scrutinized, arithmetic tests are used mostly for the purpose of controlling which of several run-time options the code will take. These tests seldom test results that were created during the run.

Enter and exit subroutine are hardware operators, and handle any address environment management that is needed. Recursive subroutines apparently are not needed, and will not be allowed.

4.4.3.4 Diagnostic Controller

The diagnostic controller is a subject for further discussion. Its instructions could be passed to it one at a time from an external channel, thus looking the same whether issued from the host or imposed by button-pushing from the maintenance panel, or they could be executed in sequence, in which case test and branch instructions are needed also. The diagnostic controller's instruction set has not yet been finalized. Control of the rest of the NSS is obtained indirectly, through the "execute (instruction)" instruction, since the CU instruction being executed can control the rest of the machine.

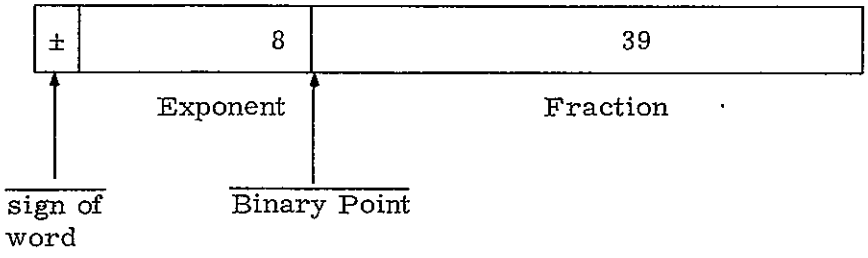
One expects that the bulk of all diagnostic and confidence programming for the NSS will be programs that run on the CU and the processors. The diagnostic controller represents a method for boot-strapping one's way up to the point where the CU is able to execute code.

Like the control unit, the diagnostic controller is a centralized control element of the synchronizable array machine. The diagnostic controller has a port to receive commands from the host, a port to emit results back to the host, and is capable of exerting control over the array even when suitable programs resident within the array are lacking.

The diagnostic controller also can receive commands from, and display results to, a maintenance panel to give the array a small degree of stand-alone maintainability.

4. 4. 4 .Data Word Formats

The standard 48-bit floating point word format is sign, 8 bits binary offset exponent, and 39 bits of fraction part. The fraction is between one half and one. Various other significances are



also coded into the exponent field.

- 11111111 (255) in the exponent field is "unrepresentable" (or "∞"), the result of exponent overflow, divide by zero, etc. Memory will normally be initialized to "-∞" with the fraction part the address.

- 00000001 (1) is "infinitesimal", the result of exponent underflow.

- 00000000 (0) prefaces zeroes, and is also used to prefix integers that have been converted to integer from floating point. Note that such integers never legally enter floating point operations without first being FLOATed, which normalizes them and attaches the proper exponent. Multiplication by zero produces a positive result.

In addition to floating point words that enter into floating point operations, some data is found in memory in half-words. Floating point packed format is the leading 24 bits of a floating point word, rounded, packed two per word. Packed integer is a word of two 24-bit integers.

The "unrepresentable" (or "infinity") code substitutes for the arithmetic fault interrupt, which is impossible to implement in any reasonable way when there are hundreds (or even thousands) of arithmetic operations being carried on at the same time.

A discussion justifying the word format is in Appendix K.

Rounding

All floating point operations are rounded except when part of programmed multiple precision operations. If we visualize a floating point number as the nearest possible respresentation, quantized, of a real number, then the unrepresented part, to the right of the end of the word, can have any value from -1/2 LSB to +1/2 LSB, with uniform distribution. When this is true, the proper procedure for rounding is to add exactly 1/2 LSB and truncate.

This scheme has an apparent bias, in that if the guard digits were exactly one half, it rounds always up, instead of randomly up or down. Actually, there are not infinitely many guard digits, just many, and the number being rounded has already been truncated ever-so-slightly, to the right of the last guard digit.

Certain details of perfect rounding are extremely difficult to implement in a lock-step machine, without paying a significant penalty either in throughput or additional logic. Here, in the SAM, we need not have all processors executing identically timed instruction, but can enjoy data-dependent timing for perfect rounding.

In addition and division rounding is done after normalization, giving about one more bit of precision than rounding before normalization. Very infrequently, the addition of 1/2 LSB will propagate a carry into the adder overflow position, causing a renormalization by one place. The extra time for this renormalization is normally bypassed, and taken only when the overflow is observed to occur.

Instruction Words

Instruction words come in two sizes, full word and half word (Figure 4-7). The full-word instruction contains a "0" prefix for "full word", an opcode field, and either two PEM addresses, or one PEM address and one or two 4-bit local register addresses, or an EM (24-bit) address and a register address. The address field is 18 bits long, with a 2-bit code for the following 4 cases for the other 16 bits.

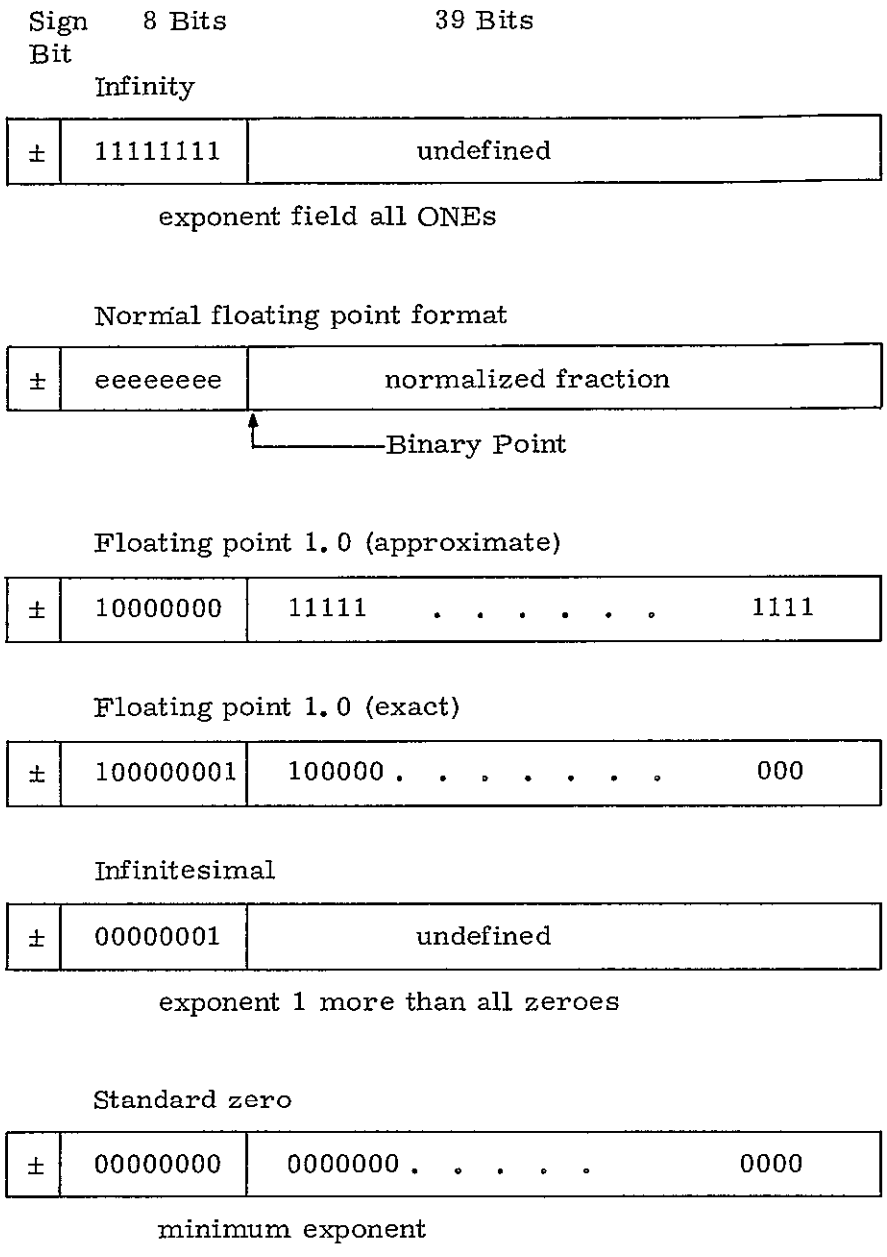


Figure 4-6. Format

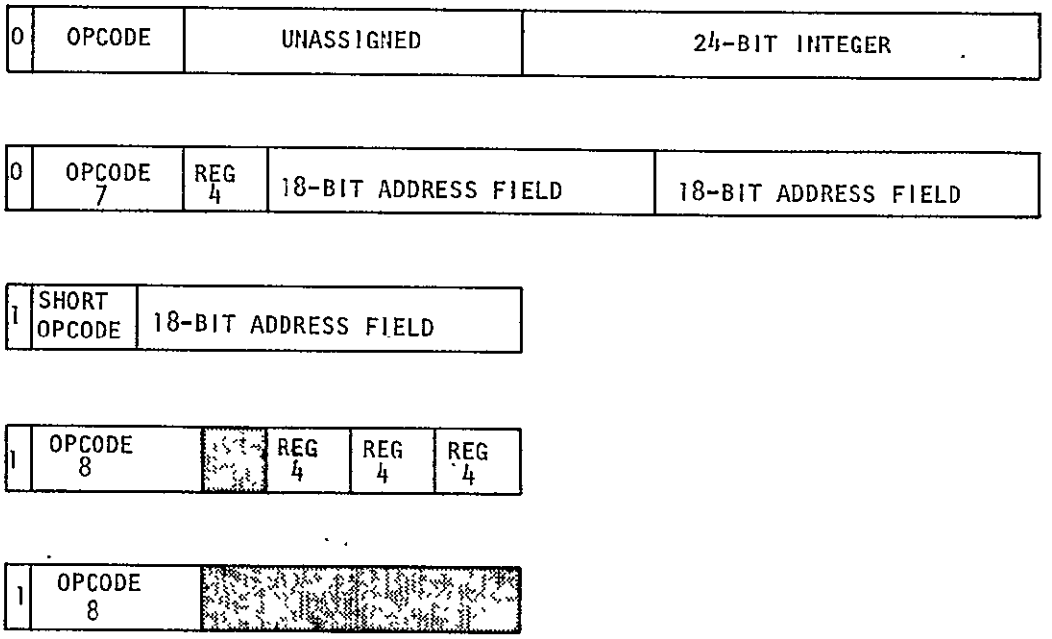


Figure 4-7a. Examples of Full-Word and Half-Word Instruction Formats

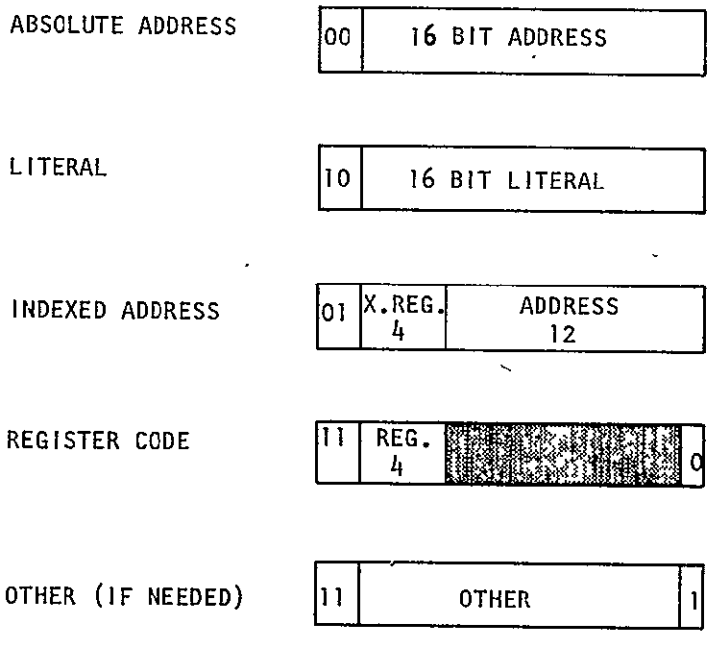


Figure 4-7b. 18-Bit Address Field Encoding

1. a 16-bit address
2. a 4-bit index register number, and a 12-bit increment to the index.
3. a 16-bit literal. Interpretation of the literal is opcode dependent.
4. a 4-bit register number or "other". 11 bits are spare.

A half-word instruction begins with "1". The other 23 bits consist of either a short (5-bit) opcode followed by an 18-bit address field, or a longer opcode followed by register addresses. Register operations have the same opcode as operations from memory, the only difference being the full-word vs. half-word bit (Figure 4-7a and b).

4.4.5 Timing

The previous sections on the PE and CU exhibit timing diagrams of some of the crucial instructions. From the timing diagrams on multiplication, multiply and add, and add in the PE section, and from the timing diagrams for EM operations in the CU section, Table 4-5 is extracted. This table gives the execution times, in the PE instruction stream, of most of the common instructions.

In addition to instruction execution times, the following information also pertains to timing.

1. The maximum instruction fetching rate is 120 ns per word. A word contains either one or two instructions.

2. The minimum instruction execution time is 40 ns for any instruction.

3. If the next instruction is overlappable with any instruction currently in execution, it will start 40 ns after that instruction or whenever it is fetched from memory. The maximum overlappability occurs at the end of a store to EM, when one can be emitting data through the TN from the byte serializer, doing a floating point operation, an integer add, while simultaneously fetching the next instruction.

4. An EM fetch cannot start until 480 ns after the end of an EM store. During this time, the PE may do any self-contained operation, such as subroutine entry or return, or calculating the address for the following EM fetch. An EM store (as seen at the PE) cannot start until 360 ns after the end of any previous EM fetch.

5. Two successive type II synchs must be separated by 180 ns. The intervening 140 ns can be used for other instructions.

Table 4-5. Timing Information (all PE times)

PE Instructions (major instructions)

Multiply	360 ns register-to-register
Add/Subtract	240 ns register-to-register
Multiply/Add	440 ns register-to-register
Divide	1800 ns register-to-register
ABS, Change Sign	40 ns register-to-register
Synchronize	180 ns Type 1 40 ns Type 2
Fetch	120 ns add to arithmetics when operand is taken from PEM
Add Integer	40 ns overlappable with floating point operations
MUL integer	240 ns last 40 ns overlappable with next floating point operations

EM Fetching and TN Operations

Fetch N words to PEM	(380 + 140 N) ns includes sending address from PE to EM. Assumes that CU instruction for setting TN (80 ns) occurred prior to PE need for instruction, and that most of the synchronization delay is overlapped with the setting of TN controls and transmission of addresses to EM. Type II synch included. PEM cycle will overlap succeeding instruction if possible. *
Fetch 1 word to PE Register	520 ns Type I synch included
Store N words from PEM	(40 + 140N) includes PEM cycle time
Store 1 word from PE Register	80 ns (type II synch)
Shift 1 Word	280 ns provided that CU is 80 ns ahead of the PEs, so that synching is overlapped
(transmit to EM data register, reset TN, and transmit back to PE)	
Read EM Module No.	40 ns Type II synch

*Addresses of the N words are to satisfy the constraint given in the "Data Allocation" section.

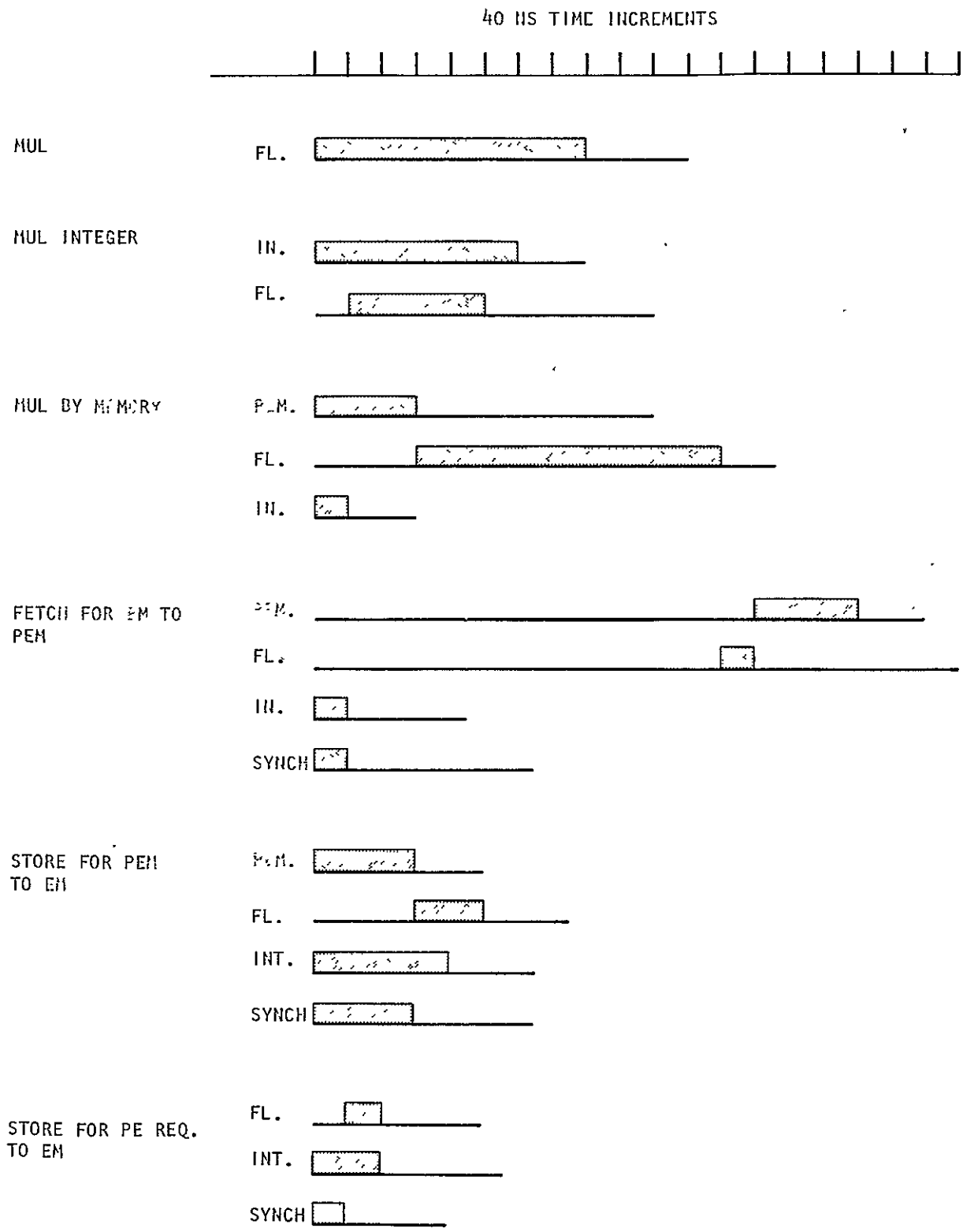


Figure 4-8. Overlappability

The PE is divided into several semi-independent parts. A given instruction will start whenever its pattern of keeping these parts busy does not interfere with the busy pattern the parts used in previous instruction. The parts are: floating point unit and registers, index arithmetic and registers, PEM, and synchronization logic. The times in Table 4-5 are a sort of "across-the-PE" time. For example, the 80 ns given to "Store to EM from PE register" actually occurs only in the integer unit, from whence comes the address. If this instruction were squeezed between two floating point operations, it might only add 40 ns to the entire operation.

Figure 4-8 shows the overlappability of a few selected instructions, to show that the settings of the counters would be in a number of cases. There are some interesting opportunities for much more overlap than assumed in Table 2-5. For example, Fetch one word from EM to PEM would be charged with only 40 ns of PE time if it could be preceded by all floating point operations and succeeded by integer operations.

4.5 DATA ALLOCATION

The transposition network will take any set of elements in extended memory, that are spaced apart uniformly in memory address, and fetch them in order to the PE's. The classical examples of vectors that are accessible thusly are rows, columns, and diagnoals of two dimensional arrays.

In fetching, in parallel, a vector of words from the EM, we need the addresses within each module of extended memory. These addresses are calculated in the processor, with processor number i calculating the address of the ith element of the vector. The compiler furnishes, to the CU, the starting module number M_O and the skip distance p. The following discussion develops the formulas whereby the processors do this address calculation.

The overall address within EM is given by:

$$A_i = A_0 + p*i$$

where A_i is the vector element address, A_0 is the address of the zeroth element of the vector, i is the element number, and p is the skip distance. This equation describes the 521 addresses of the 521 elements accessible from extended memory on a single fetch, for most common cases. One can calculate each A_i within the PE, using i = PE number. The address-within-module is given by:

$$AM_i = A_i \text{ DIV } 521$$

where DIV is a "Floor" divide, Module number $M_i = A_i$ MOD 521. DIV 521 can be made quite fast, as described in Appendix D. Note, all counts are started from zero, not one, in the above equations.

All calculations are done with algorithms supplied by the compiler; the user programmer has only sumbolic addressing to contend with.

For example, consider an array A(100, 100, 100) with indices I, J, K, which is being fetched inside DOPARALLEL I and DOPARALLEL J loops. K, the iteration number, is known. The computations go as follows·

$$Temp_1 = \text{Iteration no.} \times 100 \times 5 + \text{PE NO.}$$

$$J = Temp_1 \text{ DIV } 100 + 1$$ (assuming 5 100-long rows are being computed per crack. 512 per crack is also computable, but complicates the expressions)

$$I = Temp_1 - 100(J-1) + 1$$

$$K = \text{(Iteration no)}$$

After getting I, and J, the Address in EM is calculated

$$A_i = A_0 + 100 \times 100 \times (K-1) + 100 \times (J-1) + I$$

Finally the address within module is given by

$$AM_i = A_i \text{ DIV } 521$$

As appendix J shows, the DIV operator can be more economically implemented if M_i is read from the EM module which will by this time be attached to the processor, giving the equation

$$AM_i = (A_i - M_i) \text{ DIV } 521$$

The above equations are an upper bound on address computation complexity. Good compilers add increments to addresses, rather than adding iteration-number times a single increment to the base each time, thus saving multiplies. Furthermore, the J and I calculations are done only once per outer iteration. A fully coded out example appears in Appendix A.

Equivalencing inside EM ARRAYS is restricted because of the need to perform EM fetches and stores efficiently. For example, in

```
DO N=1, 18
B(N, xx) = A(N, xx, J, K) (xx is any other index, irrelevant here)
```

(inside DOPARALLEL J and DOPARALLEL K) we have declared the first index of B to be the one that increments by one, in going through the PEM addresses, in accordance with FORTRAN custom. In EM, however, the 18 elements of A, for the 18 neighboring addresses in B, are in the same module, not neighboring addresses. For interlacing, they are separated in the EM module by an odd increment. Therefore, the first index in EM arrays is the index with the largest increment, and the compiler will pad out the space assigned to the other indices in an EM array to occupy an odd multiple of 521 words. For example, consider A(18, 100, 100, 100). This will appear in EM as eighteen successive areas, each containing 1,000,000 words of data, and occupying 1,000,841 words of space (521 x 1921, where 1921 is the smallest odd number equal to or larger that 1,000,000/521). The increment to the EM address-within-module is 1921 for each word fetched in the burst of 18. This rearrangement of EM arrays is invisible to the programmer.

It is important that these same arrays appear on the outside, in the B 7800 programming environment, in conformance with their declaration in NSS FORTRAN. A (5, 100, 100, 100) in EM, when transferred to the B 7800 environment, can be addressed as an array AA(5, 100, 100, 100). Hence the DBM to EM transfer path must be supplied with the correct instructions to rearrange the array coming from DBM into the format used in EM. This is a hardware instruction, and will be invisible to the user programmer. The sequence of loading A(5, 100, 100, 100) from DBM to EM; where the last three indices are known to be the geometry

indices, is therefore as follows. The first five elements are loaded from successive addresses in DBM into five addresses in EM module 0 at increments of 1921; the next five into EM module 1 at increments of 1921; and so on through 521 EM modules. The base address is now incremented by 1 and the process continues.

Thus, the declaration of an EM array inside the NSS is identical to the declaration of that same array outside the NSS. The user programmer never sees the EM hardware address, just as he never sees that there are 512 processors.

Further details of EM data allocation are described with respect to this hardware address. Further study of some additional cases is called for in phase II of the project.

Both one-dimensional vectors, and many two-dimensioned data objects have the property that the successive module numbers M_i of their elements are separated by a constant amount

$$M_i = (M_0 + i * p) \text{ MOD } 521$$

where p is that separation

The transposition network takes such a scrambled sequence of memory modules and orders the data on the PE side so that ith element of the vector appears at the ith PE. Because there are more PE's than the extent of any single dimension of the array, the transposition network must be able to take two-dimensional subsets of three dimensional arrays, and distribute them in parallel across the entire set of PE's. Furthermore, it will usually be true that the two-dimensional subset is only part of the entire extent of the array. For example, suppose there is a computational grid of 100 x 100 x 100, and 512 PE's. The programmer has nested two looped statements, DO K=1, 100 and DO L=1, 100, specifying a 10, 000 element wide parallelism, which the compiler then must cut up into pieces.

For the case of array A(100, 100, 100), the ten thousand elements of A for a fixed L, (A(*, *, L)), are all packed with separation unity, so the compiler can cut the 10, 000 wide parallelism into pieces of 5 x 100 (or, if row ends don't matter,

into pieces 512 long, although there will still be 20 of them). The ten thousand elements of A(J*, *)* are separated by a separation of 100. Again, the 512 successive elements of a 100-ordered vector are easily handled by the transposition network. For A(*, K, *), there are 100 rows of 100 elements each. The beginning module number of the second row is just 101 after the beginning module number of the first, because 10,000 modulo 512 = 101. Five 100-long rows fit in parallel across the 521 modules with starting points just 101 apart.

At first blush, this appears to be a lucky accident, allowing five successive rows of 100 to fit end-to-end along the 521 memory modules. A few numerical experiments show that this is no lucky accident, that almost all two-dimensional subsets of the form A(*, K, *), where the middle index is not the parallel one, work well. The conditions are these:

> The problem is to find a number of rows (number =521/JMAX) that start at positions that are equal to, or just over JMAX apart in their starting positions.
>
> Let N = 521/JMAX, and G and H arbitrary integers. Then we wish to satisfy
>
> (H x JMAX x KMAX) modulo 521 = (JMAX + G) modulo 521
>
> where both H and G are adjustable parameters. In fact, all that is required is that reasonable solutions be available for a variety of values of JMAX and KMAX, not for all of them. The numerical experiments show that reasonable two-dimensional fetching, simultaneous on first and third index, is available for most combinations of JMAX and KMAX.
>
> The compiler will emit "WARNING: ARRAY SIZE CAUSES INEFFICIENT EM FETCH PATTERN" when bad combinations are met during compilation.

* "*" is used to designate items fetched in parallel.

GLOSSARY

Burroughs Scientific Processor (BSP)	- An example of a lockstep array.
Concurrency	- The execution of program, which may be the same or different, on more than one data set simultaneously.
Control Unit (CU)	- There is one control unit per SAM. It coordinates the actions of the processors, and executes the portion of the operating system that is local to the SAM. It includes Control Unit Memory.
Control Unit Memory (CUM)	- A random access memory for the control unit's program and data.
Data Base Memory (DBM)	- A large memory used for staging jobs for the SAM. It contains the input files for the next job, and the output files for the last job.
Diagnostic Controller (DC)	- A controller which exerts maintenance type control over the CU and the processors. It can also be used for initialization.
Extended Memory (EM)	- A set of 521 memory modules, being the memory containing the main data base of the program.
Single Error Correction Double Error Detection (SECDED)	- An error correction code.
Navier Stokes Solver (NSS)	- That particular SAM that is part of the NASF.
Numeral Aerodynamic Simulation Facility (NASF)	-
Parallelism	- Concurrency in which the identical program is being executed simultaneously on different data.

Processor Element (PE)	- The logic engine of the processor.
Processor	- One of 512 cooperating processors all executing the same program.
Processing Element Memory (PEM)	- The data memory of the processor.
Processing Element Program Memory (PEPM)	- The program memory of the processor.
Synchronizable Array Machine (SAM)	- A particular architecture of parallel processor. The NSS is a SAM.
Temporary Propagation	- The tendency to require more temporary variables in a parallel machine than are required for the same algorithm programmed for a serial machine.
Transposition Network	- A set of switchable data paths between the EM modules and the processors.

REFERENCES

1. J. L. Steger "Implicit Finite Difference Simulation of Flow About Arbitrary Geometries with Application to Airfoils", submitted to ATAA, 1977.

2. Joseph E. Wirsching "Computer of the 1980's – Is It An Array of μComputers?" WESCON 75

3. R. M. Beam and R. F. Warming "An Implicit Finite-Difference Algorithm for Hyperbolic Systems in Conservation-Law Form".

4. R. W. MacCormack and B. S. Baldwin, "A Numerical Method of Solving the Navier-Stokes Equations with Application to Shock-Boundary-Layer Interactions" AIAA 13th Aerospace Sciences Meeting, Jan. 1975.

5. R. W. MacCormack "An Efficient Numerical Method for Solving the Time-Dependent Compressible Navier-Stokes Equations at High Reynolds Number" NASA Technical Memorandum, Ames Research Center, July 1976.

6. Steger, J. L. and P. Kutler, "Implicit Finite Difference Procedures for the Computation of Vortex Wakes" AIAA paper 76-385, 1976.

7. Shyh-Ching Chen and David J. Kuck, "Time and Parallel Processor Bounds for Linear Recurrence Systems" IEEE Trans Computers, Vol. c-24 No. 7, July 75.

8. Reigel, Earl "Parallelism Exposure and Exploitation in Digital Computing Systems" (either) Burroughs Corporation, FSSG, TR-69-4 (or) Doctor's Thesis, University of Pennsylvania, 1969.

9. Chin Chae and Herschel H. Loomis Jr., "High Rate Realization of Finite-State Machines" IEEE Trans Computers, Vol. C-24, No. 7, July 75.

10. P. Burnick and David J. Kuck "The Organization and Use of Parallel Memories", IEEE Trans Computers, Dec. 71, prefers N price and just larger the 2^n (17, 67, 257).

11. David J. Kuck "A Survey of Parallel Machine Organization and Programming", ACM Computing Surveys, Vol. 9, No. 1, March 1977

12. D. J. Farber and K. C. Larson, "The System Architecture of the Distributed Computer System - - The Communications System", Proc. Symp. on Computer-Communication Networks and Teletraffic, Polytechnic Institute of Brooklyn, 1972.

13. R. J. Swan, A. Bechtolsheim, Kwok-Woon Lai and J. K. Ousterhout, "The Implementation of the Cm* Multi-Microprocessor", Cargenie-Mellon preprint, Nov. 1976 (submitted to 1977 National Computer Conference).

14. C. G. Bell, R. C. Chen, S. H. Fuller, J. Crason, S. Rege, and D. P. Siewiorek, "The Architecture and Applications of Computer Modules: A Set of Components for Digital Design", IEEE Computer Society International Conference, Comp. Con. 73, March 1975, pp. 177-180.

15. May 1977 "Computer" Special Issue on Stack Architecture.

16. Roger C. Swanson, "Interconnection for Parallel Memories to Unscramble p-ordered Vectors", Nov. 1974, IEEE Trans. Computers.

17. J. Bruce Mawson, "An EAI Discussion of an Advanced Hybrid Computer System", Electronic Associates, Inc., West Long Branch, N. J.

18. Spectrum, March 1977, Van Tuyl and Liechti, "Gallium Arsenide Spawns Speed" (MESFETs)

19. Electronics, August 5, 1976, special issue on optical communications.

20. D. Heller, "A Survey of Parallel Algorithms in Numerical Linear Algebra" February 1976, Carnegie-Mellon preprint.

21. AT&T 1977 annual report (short description of the Chicago experiment, in which customer traffic is being carried over optical waveguide)

22. Analog Devices' Product guide (for state-of-the-art analog components).

23. Analog dialog, Vol. 11, Nov. 1, 1977, (monolithic multiplier, 18-bit D-A).

24. IEEE Journal of Solid State Circuits, April 1977. Special issue on I^2L.

25. G. H. Barnes, R. M. Brown, M. Kato, D. J. Kuck, D. L. Slotnick and R. A. Stokes, "The ILLIAC IV Computer", IEEE Trans. Computers, Vol. C-17, No. 8, August 1968.

26, Harry Gray, "Digital Systems Engineering", Chapter 9

27. Daniel Shanks "Solved and Unsolved Problems in Number Theory", Spartan Books, 1962.

28. W. H. Dunn, C. Eldert, P. V. Levonian "A Digital Computer for Use in an Operational Flight Trainer", I. R. E. Transactions on Electronic Computers, June, 1955, pp. 58-60.